

## REPORT DOCUMENTATION PAGE

Form Approved  
by 0704-0188

2

DTIC FILE COPY

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing existing data sources, gathering new data, searching existing data sources, gathering other aspect of this collection of information, and completing and reviewing the collection of information, including suggestions for reducing this burden, to Washington Headquarters, VA 22202-4302, and to the Office of Management and Budget, Paperwork Project, Washington, DC 20503.

AD-A217 154

4. searching existing data sources, gathering other aspect of this collection of information, and completing and reviewing the collection of information, including suggestions for reducing this burden, to Washington Headquarters, VA 22202-4302, and to the Office of Management and Budget, Paperwork Project, Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE August 1981		3. REPORT TYPE AND DATE COVERED Technical Note--Oct 79-Aug 81	
4. TITLE AND SUBTITLE Project STEAMER: III. Using Qualitative Simulation to Generate Explanations of How to Operate Complex Physical Devices				5. FUNDING NUMBERS 0603720N Z1177-PN Z1177-PN.03	
6. AUTHOR(S) Kenneth Forbus, Albert Stevens, Bolt Beranek & Newman, Inc.				8. PERFORMING ORGANIZATION REPORT NUMBER NPRDC-TN-81-25	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Navy Personnel Research and Development Center San Diego, California 92152-6800				10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Chief of Naval Operations (OP-01), Navy Department, Washington, DC 20350-2000					
11. SUPPLEMENTARY NOTES					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The main objective of the STEAMER effort is to develop and evaluate advanced knowledge-based techniques for use in low-cost portable training systems. The project is focused on propulsion engineering as a domain in which to investigate these computer-based training techniques.  This report, the third in a series on the STEAMER project, describes a method based on incremental qualitative simulations for automatically generating explanations and animating diagrams to explain the operation of complex physical devices such as those found in propulsion plants.					
14. SUBJECT TERMS Propulsion engineering, intelligent computer-assisted instruction, artificial intelligence, computer graphics				15. NUMBER OF PAGES 71	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED		18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED		19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	
				20. LIMITATION OF ABSTRACT UNLIMITED	

NPDC TN 81-25

AUGUST 1981

**PROJECT STEAMER: III. USING QUALITATIVE  
SIMULATION TO GENERATE EXPLANATIONS OF  
HOW TO OPERATE COMPLEX PHYSICAL DEVICES**



**NAVY PERSONNEL RESEARCH  
AND  
DEVELOPMENT CENTER  
San Diego, California 92152**



NPDC / 1N 81-25

C.2

**PROJECT STEAMER: III. USING QUALITATIVE SIMULATION  
TO GENERATE EXPLANATIONS OF HOW TO OPERATE  
COMPLEX PHYSICAL DEVICES**

Kenneth Forbus  
Albert Stevens

Bolt Beranek and Newman, Inc.  
Cambridge, MA 02138

Reviewed by  
John D. Ford, Jr.

Released by  
James F. Kelly, Jr.  
Commanding Officer



Accession For	
NTIS GRA&I	N
DTIC TAB	U
Unannounced	U
Justification	
By	
Distribution /	
Availability Codes	
Dist	Availability Codes
A-1	

Navy Personnel Research and Development Center  
San Diego, California 92152

## FOREWORD

This research and development was conducted under contract N00123-81-D0794 with Bolt Berenek and Newman, Inc. in support of Navy Decision Coordinating Paper Z1177-PN (Advanced Computer-Aided Instruction), subproject Z1177-PN.03 (STEAMER: Advanced Computer-Based Training for Propulsion and Problem Solving). It was sponsored by the Chief of Naval Operations (OP-01). The main objective of the STEAMER effort is to develop and evaluate advanced knowledge-based techniques for use in low-cost portable training systems. The project is focused on propulsion engineering as a domain in which to investigate these computer-based training techniques.

This report, the third in a series on the STEAMER project, describes a method based on incremental qualitative simulations for automatically generating explanations and animating diagrams to explain the operation of complex physical devices such as those found in propulsion plants. Previous reports described an initial framework for developing techniques for automatically generating explanations of how to operate complex physical devices and provided a user's manual for the STEAMER interactive graphics package (NPRDC TNs 81-21 and 81-22). Intended users of these reports are system maintainers and other research personnel.

Appreciation is extended to the personnel at the Surface Warfare Officers School at Newport, Rhode Island, who provided information about the 19E22 simulator and participated in several beneficial discussions about the nature of the training problem being addressed in this R&D effort.

The contracting officer's technical representative was Dr. James D. Hollan.

JAMES F. KELLY, JR.  
Commanding Officer

JAMES J. REGAN  
Technical Director

## SUMMARY

### Problem

The main objective of the STEAMER effort is to develop and evaluate advanced knowledge-based techniques for use in low-cost portable training systems. The project is focused on propulsion engineering as a domain in which to investigate these computer-based training techniques. One of the important problems of the STEAMER system is generating explanations about the operation of propulsion plant components and sub-systems.

### Objective

This report describes a method, based on incremented qualitative simulations, for automatically generating explanations and animating diagrams to explain the operation of complex devices.

### Results

The method uses a technique based on incremental qualitative simulation. An incremental qualitative simulation models the changes that each component of the device undergoes in response to an external change. The report first describes incremental qualitative simulation, showing how it may be applied to the propulsion engineering domain. This is followed by a description of how it is augmented to provide automatically-generated explanations of the operation of complex physical devices.

### Conclusions

It is possible to generate coherent, understandable explanations of the operation of physical devices from a qualitative simulation of the device operation. To do this, a general system for generating explanations and animating diagrams, based on incremental qualitative simulation, has been implemented. New types of devices can easily be added to this system. These techniques make possible learning environments in which students can experiment with complex devices and see explanations of the effects of various changes. These techniques are currently being integrated as part of the STEAMER system.

## CONTENTS

	Page
INTRODUCTION . . . . .	1
Problem . . . . .	1
Purpose . . . . .	1
QUALITATIVE MODELS . . . . .	1
RESULTS . . . . .	3
An Example Explanation . . . . .	3
Incremental Qualitative Simulation . . . . .	3
Generating Explanations . . . . .	11
CONCLUSIONS . . . . .	14
REFERENCES . . . . .	17
APPENDIX--CONLAN: THE CONSTRAINT INTERPRETER . . . . .	A-0

## LIST OF FIGURES

1. Successive frames of the explanation generated for a spring-loaded reducing valve . . . . .	4
2. The component models currently implemented . . . . .	9
3. The grammar that adds punctuation and turns phrases into sentences and paragraphs . . . . .	13

## INTRODUCTION

### Problem

The STEAMER project is an attempt to develop advanced computer-aided instruction systems for teaching the operation of Navy propulsion plants. Before students can understand how a propulsion plant works, they must be knowledgeable in basic physics and thermodynamics, understand how many different types of devices work, and be familiar with overall plant organization and interaction.

### Purpose

The purpose of this report was to describe a method for automatically generating explanations and animating diagrams to explain the operation of complex physical devices such as those found in propulsion plants.

## QUALITATIVE MODELS

Initial ideas in this domain centered on using a mathematical model of a propulsion plant as a basis for the computer-assisted instruction (CAI) system. A typical math model represents the time varying behavior of selected attributes of the different objects and substances in the device being modelled. An explanation generated using such a model would "show" (visually or verbally) the device in successive temporal states. In each succeeding state, the attributes being illustrated would all change a small amount and thus one would see relatively continuous changes of many parts of the device. Although this type of model is fine for illustrating the device's detailed behavior, it is not a good basis for providing an explanation or description that a person can easily grasp. There is a growing amount of evidence that human understanding of physical systems is based on qualitative models of those systems. This evidence comes from psychological studies (Larkin, McDermott, Simon, & Simon, 1980; Stevens, Collins, & Goldin, 1979) and is supported by successes in artificial intelligence in actually constructing systems that reason about physical situations using qualitative models (deKleer, 1979a, Forbus, 1980).

Consider the following explanation of an air-operated pilot valve:

As the controlled pressure (discharge pressure from the diaphragm control valve) increases, increased pressure would be applied to the diaphragm of the direct acting control pilot. The valve stem would be pushed down and the valve in the control pilot would be opened, thus sending an increased amount of operating air pressure from the control pilot to the top of the diaphragm control valve. The increased operating air pressure acting on the diaphragm of the valve would push the stem down and--since this is an upward seating valve--this action would open the diaphragm control valve still wider. (Bureau of Naval Personnel, 1970), p. 383.

This explanation is comprised of a set of events, each describing a qualitative change in some part of the device. The explanation is linearized and describes how physical effect is passed from one component to another. It ignores the true temporal changes: those things that are happening are happening continuously and simultaneously. The discrete and ordered nature of the events in the explanation is imposed.

If a tutor that can instruct students about the *fundamentals* of propulsion plants is to be built, the descriptions used by the tutor should support explanations like that of the pilot valve. The next section describes how such explanations can be generated automatically by using an incremental qualitative simulation of a subsystem. First, an explanation generated using such a technique is shown. Then incremental qualitative simulation as developed by deKleer (1979b) is described, and it is shown how the technique may be applied to the propulsion engineering domain. Finally, the section describes how the technique is augmented to provide automatically generated explanations of the operation of complex physical devices.

## RESULTS

### An Example Explanation

Figure 1 presents an explanation generated using an underlying qualitative simulation. Each panel of the explanation is drawn from the actual computer display that a student sees. Successive panels denote successive states of the display. (These have been redrawn rather than photographed because of printing constraints: the original displays are in color and are animated with attention-focussing blinking.) The device described is a spring-loaded reducing valve, a common type of control device that serves to supply steam at a constant reduced pressure to a set of varying loads.

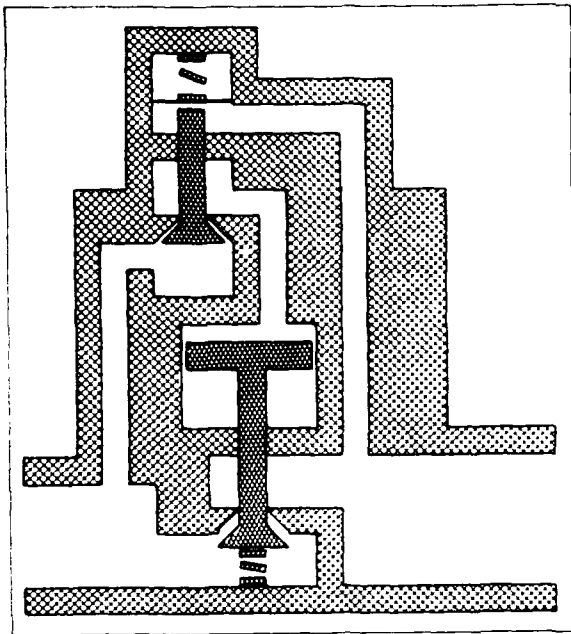
It is instructive at this point to read the explanation. The first command, "Consider device spring-reducer-valve," indicates which device is to be studied. The controlled parameter is chosen to be the pressure in the output port by the command "Consider (>> press Output-Port V1)."<sup>1</sup> The explanation is generated in response to the command "Suppose U." The system qualitatively simulates the effects of the pressure increase. By analyzing this qualitative simulation, the system discovers that the valve exhibits negative feedback. The student then asks for further clarification by typing "Explain." The system saves enough information during the qualitative simulation to reconstruct the sequence of events that led it to that conclusion. It is this saved event description that is turned into readable English and graphics and presented to the student one event at a time.

### Incremental Qualitative Simulation

The basic idea for a qualitative simulation comes from the observation that, when trying to understand or explain a device, people often use a description of how parts of it change when some influence is applied to the system. The description of the control pilot operation presented previously is an example of such an explanation. The changes in

---

<sup>1</sup>The input mechanism to the system has not yet been satisfactorily human-engineered. Ideally, the student would simply point at the part of the device he was interested in and denote the parameter to change.

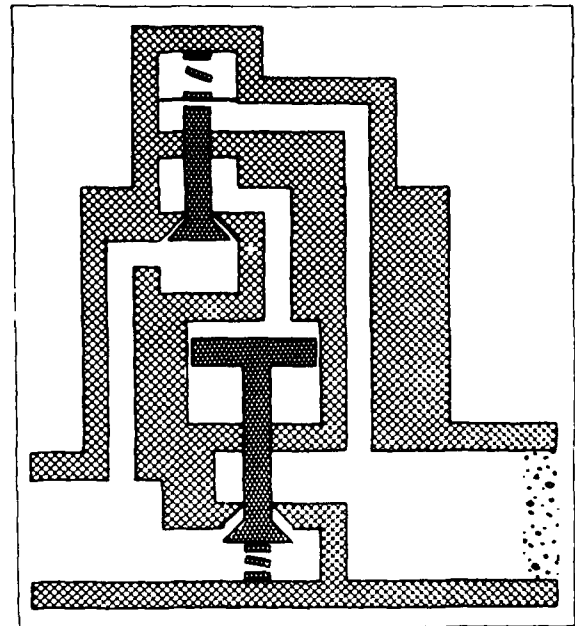


1. → CONSIDER DEVICE SPRING-REDUCER-VALUE  
WHAT SHOULD IT BE CALLED?

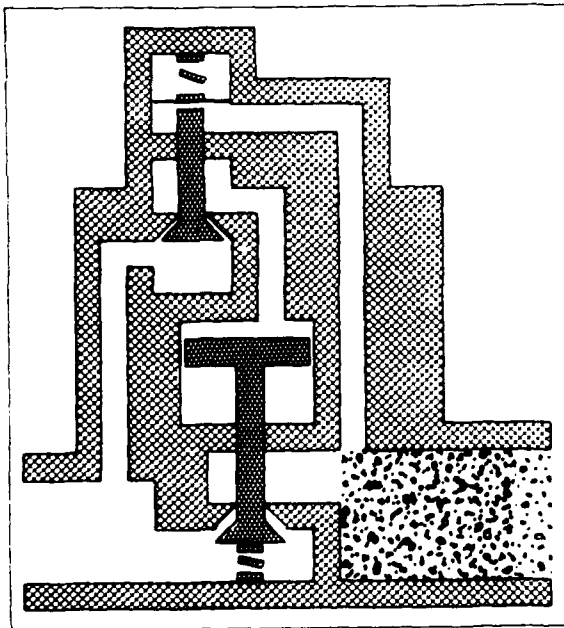
V1  
OK

→ CONSIDER (>> PRESS OUTPUT-PORT V1)  
OK

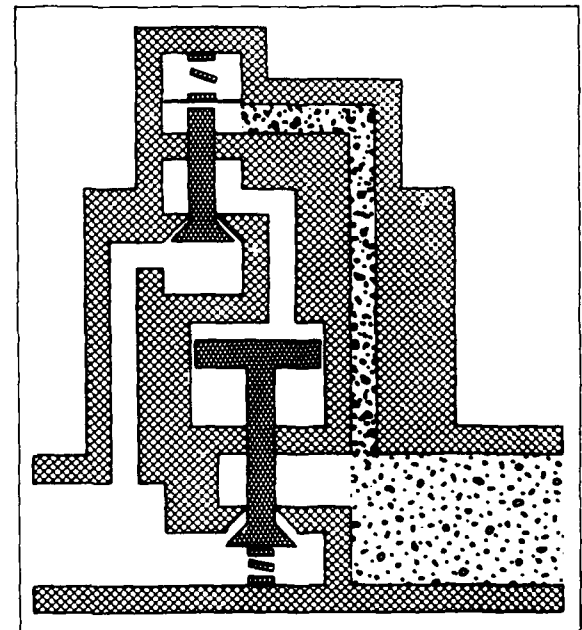
→ SUPPOSE U  
WHEN THE VALUE IS INCREASING,  
THE SYSTEM TRIES TO MAKE IT DECREASE.  
THIS MEANS IT HAS NEGATIVE-FEEDBACK.



2. EXPLAIN  
WHEN THE PRESSURE IN THE OUTPUT PORT  
RISES,

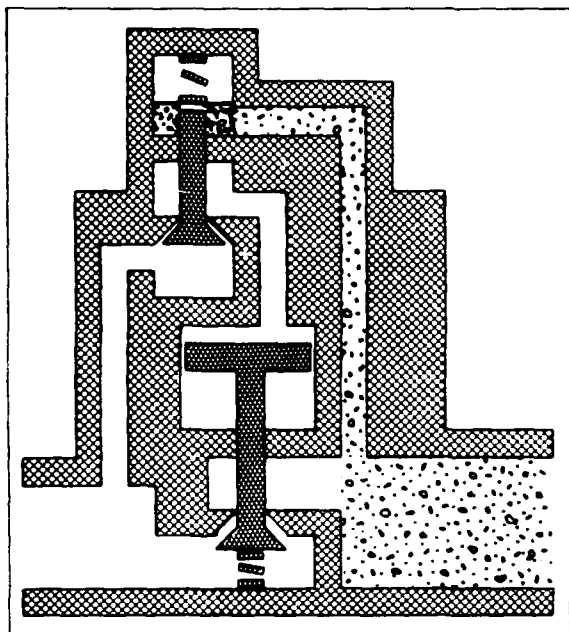


3. THE PRESSURE IN CHAMBER 0 RISES.

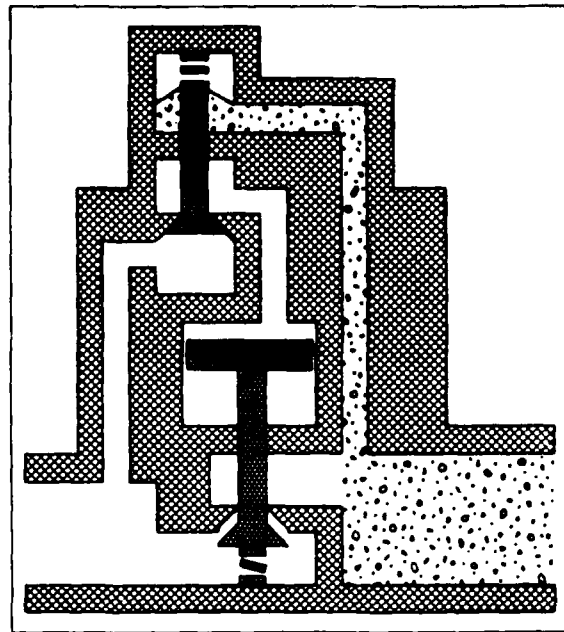


4. THE PRESSURE IN THE LOW PRESSURE PORT RISES.

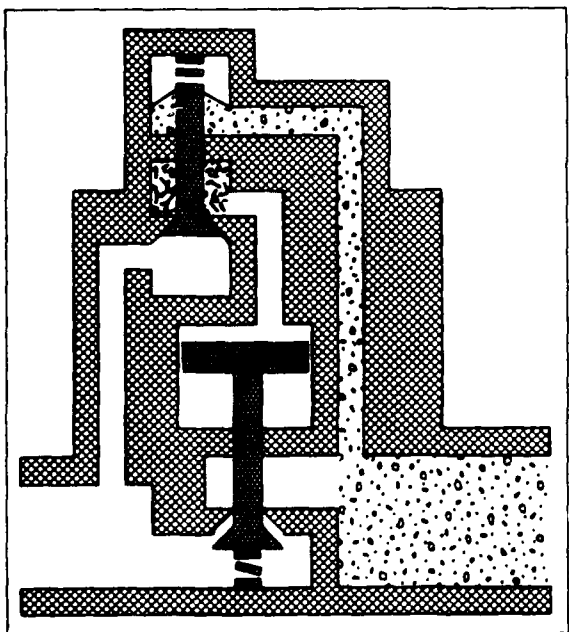
Figure 1. Successive frames of the explanation generated for a spring-loaded reducing valve.



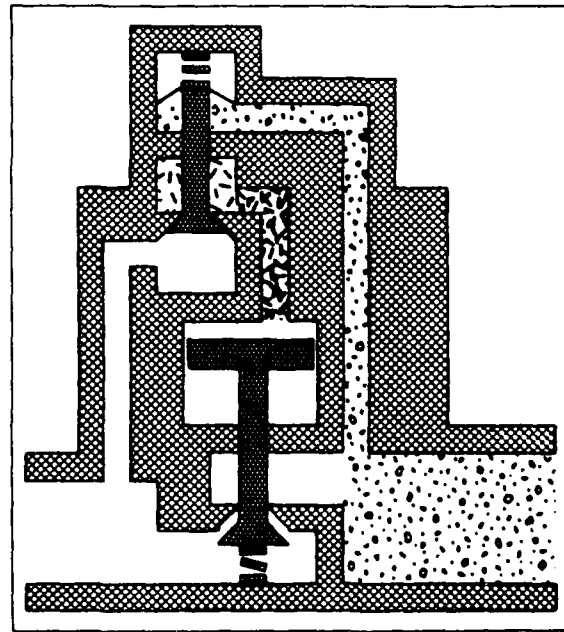
5. AND THE PRESSURE IN CHAMBER 5 RISES.



6. THE INCREASING PRESSURE PUSHES THE DIAPHRAM UP AND CLOSES THE AUX VALVE.

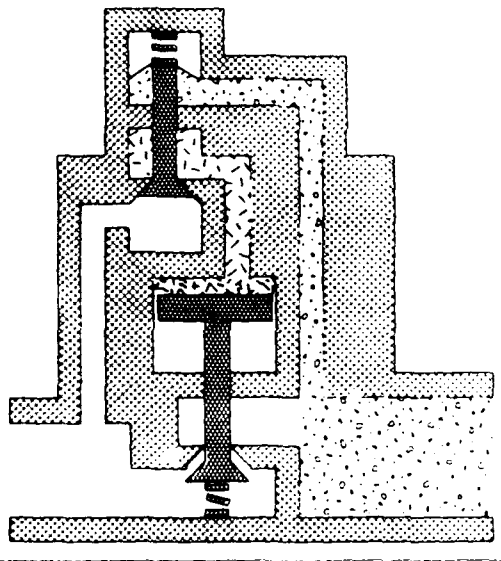


7. THE PRESSURE IN THE AUX VALVE'S OUTPUT SIDE FALLS,

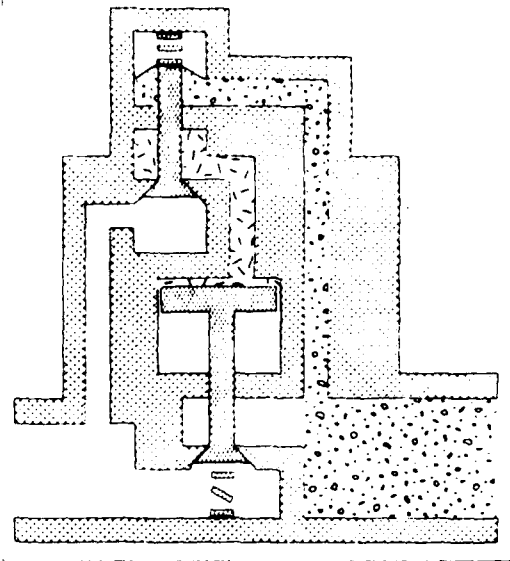


8. THE PRESSURE IN THE PISTON STEAM PORT FALLS,

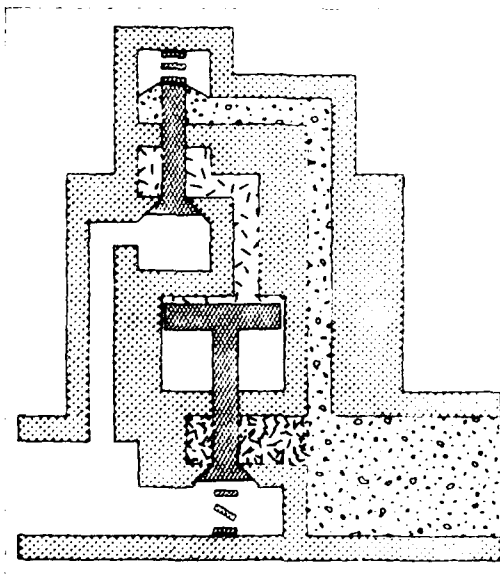
Figure 1. (Continued).



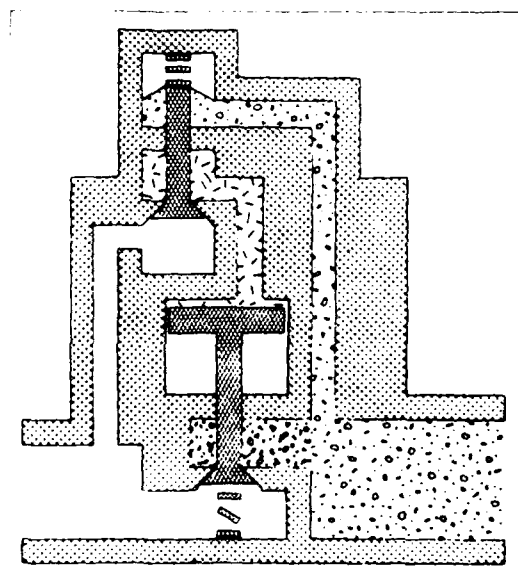
9. AND THE PRESSURE IN CHAMBER 4 FALLS.



10. THE PISTON MOVES UP AND CLOSES THE MAIN VALVE.



11. THE PRESSURE IN THE MAIN VALVE'S OUTPUT SIDE FALLS.



12. ALSO WHEN THE PRESSURE IN CHAMBER 0 RISES THE PRESSURE IN THE MAIN VALVE'S OUTPUT SIDE RISES.  
NOTICE THAT WHEN THE PRESSURE IN THE OUTPUT PORT RISES IT MEANS THE PRESSURE IN THE MAIN VALVE'S OUTPUT SIDE RISES. HOWEVER, IT ALSO CAUSES THE SYSTEM TO ACT SO THAT THE PRESSURE IN THE MAIN VALVE'S OUTPUT SIDE FALLS. THIS MEANS IT EXHIBITS NEGATIVE FEEDBACK.

Figure 1. (Continued).

physical quantities such as pressure or the position of a valve are typically described by using the sign of the derivative of the change. Thus, for a pressure, the changes are "up," "down," or "constant." The sequence of events in such a simulation depends on how the components of the device are connected together; changes in one quantity can affect only those other quantities related to it through some sort of connection. These ideas are the essence of the incremental qualitative (IQ) analysis formalized by deKleer for electronic circuits.

The power of IQ simulation comes from the local nature of the component interactions. Using a small number of different types of component models that describe how physical quantities interact locally, a complex device can be modelled by specifying the connectivity of the component devices. Once certain assumptions about the operation of the device have been made, propagating a physical effect can be done for each component model independently from the rest of the system. Thus, the effects of changes to input quantities can be easily traced through devices of arbitrary complexity.

deKleer's analysis was based on a four-element algebra to represent changes of electrical quantities (C for constant, U for increasing, D for decreasing, and ? for indeterminate). Circuit components were modelled with rules that described how changes in an electrical quantity affected other quantities within the component. For example, a resistor would be modelled with the following rules:

1. Voltage change ==> current change.
2. Current change ==> voltage change.

Thus, during the simulation, if the voltage was U, the current would be U and the effects of this change would be propagated to adjacent devices.

In the domain of propulsion engineering, the quantities of concern are things like pressures, valve openings, and flows. The algebra of change used is the same as deKleer's. The component models used so far are very simple. Spaces in the device are modelled by chambers, with ports and pipes transmitting pressure changes through them. Valves are

modelled in terms of changes in their openings. When the valve opening increases, the pressure in the input side decreases and the pressure in the output side increases. When the valve shuts, the opposite happens. A translator models collections of components that turn the change in one type of quantity into another (such as the diaphragm/spring/valve stem combination that causes a change in pressure to change the position of a valve). An example of the rules for one of these devices, a single-port chamber, is:

1. Port pressure ==> chamber pressure.
2. Chamber pressure ==> port pressure.

Thus, if the pressure at the port increases, the pressure in the chamber will increase. The component models that have been implemented and their rules are listed in Figure 2.

In deKleer's work, the focus was on using the IQ analysis of an electrical circuit to recognize its function (such as amplifier or comparator). This required that all the various assumptions about the state of components be made (e.g., whether a diode was in its normal operating region or was saturated) and then the correct interpretation found by using teleology to prune the alternatives. So far in the present work, assumptions have been required only for the valve and translator models. For a valve, it is assumed that a direction of flow through that valve establishes one side as the input and the other as the output. To reduce the complexity of the computations in the tutorial setting, the assumptions necessary for correct operation can be specified in advance. The assumptions are made explicit rather than implicit to allow the student to observe the effects of changing them.

The descriptions are expressed in the constraint language CONLAN, which is described by Steele and Sussman (1978) and in the appendix. A qualitative simulation of a device is obtained by simply specifying a value from the IQ algebra for a selected part of the device (such as the output port for the spring reducer valve) and running the constraint interpreter on it. The interpreter deduces values for as many of the component quantities as it can by running the rules associated with the component models. It records the results of this qualitative simulation as a graph of the quantities connected by the

The conventions are:

- (1)  $\langle a \rangle \Rightarrow \langle b \rangle$  means "When  $\langle a \rangle$  is known, set  $\langle b \rangle$  to it".
- (2)  $\langle a \rangle == \langle b \rangle$  is equivalent to  $\langle a \rangle \Rightarrow \langle b \rangle$  and  $\langle b \rangle \Rightarrow \langle a \rangle$ .
- (3) Opposite(value) means "If value=D then U, else if value=U then D, else value".

One Port Chamber

Port pressure == Chamber pressure

Two Port Chamber

```
Port1 pressure == Port2 pressure
Port1 pressure == Chamber pressure
```

Three Port Chamber

```
Port1 pressure == Port2 pressure
Port2 pressure == Port3 pressure
Port1 pressure == Chamber pressure
```

Pipe

End1 pressure == End2 pressure

## Continuous Valve

```

If valve open then opening ==> input pressure down
                        ==> output pressure up
                        closing ==> input pressure up
                        ==> output pressure down
else opening ==> valve open

```

(This assumes a non-zero flow to determine which side is the input and which the output.)

## Translator

```

    If invert?=NO then input == output
    else Opposite(input) == output.

```

**Figure 2. The component models currently implemented.**

rules used to deduce them. The graph describing the history of the simulation is used as the basis for generating an explanation.

The program that has been built uses the explanations it generates to illustrate the principles of feedback. This requires some extra analysis in addition to the qualitative simulation. Specifically, it requires choosing some parameter as the controlled parameter, and interpreting the response of the device to changes in this parameter accordingly. For example, the controlled parameter in the spring-loaded reducing valve is the pressure on the output side. It is the response of the device to changes in this pressure that is of primary interest.

The system currently is capable of recognizing and explaining instances of negative and positive feedback, as well as stable, unstable, and open-loop systems. Recognition of the stability and type of feedback depends on two types of events that can occur within the constraint interpreter: clashes and coincidences. A clash occurs if some rule tries to set a quantity to a value different than a value obtained by another means. A coincidence occurs if a rule tries to set a quantity to the same value obtained by another means.

Negative feedback is indicated by the constraint interpreter detecting a clash involving the controlled variable. For example, if the controlled parameter is a pressure that increases and the interpreter deduces that this results in that pressure decreasing, then it is considered that the device exhibits negative feedback. This is the case with the reducing valve model; output pressure changes, either up or down, result in the deduction that the output pressure will go the other way.

Positive feedback is indicated by the constraint interpreter detecting a coincidence involving a controlled variable that is changing. For example, if the controlled parameter was a pressure increase and the interpreter deduces that this results in a pressure increase of the same quantity, then it is considered that the device exhibits positive feedback.

A device is considered stable if the controlled parameter is constant and the system detects a coincidence. It is considered unstable if the controlled parameter is constant

and the system detects a clash. An open loop device is indicated if there are no clashes or coincidences.<sup>2</sup>

### Generating Explanations

To be useful in a tutorial setting, a qualitative simulation of a device must be turned into an understandable explanation. Using a qualitative simulation simplifies this problem because the events deduced by the simulation are similar to those that appear to be naturally used by people. However, the terms used to communicate the results of the simulation must also match those a student would use and be in an understandable form. In this section, we discuss how we turn the simulation into an English explanation, augmented with an animated diagram of the device.

The transformation of the paths of computation in the constraint network into an explanation consists of three steps. First, the graph is transformed into a string of events. Each event is a pair of a rule and the quantity it sets. Explicit markers are added to indicate events that cause several others and places where several events combine to cause a single event.

The second step in the processing consists of pruning the event string until only those events that give rise to English phrases remain. This is accomplished easily because, associated with the device, is a table of functions accessed by the name of a rule or quantity. Each of these functions returns an appropriate phrase when run on the rule or quantity it is associated with. For example, the function associated with the pressure in the Output Port in the spring reducer valve returns the string "the pressure in the Output Port" and either "rises," "falls," or "endures," depending on whether the value in the part

---

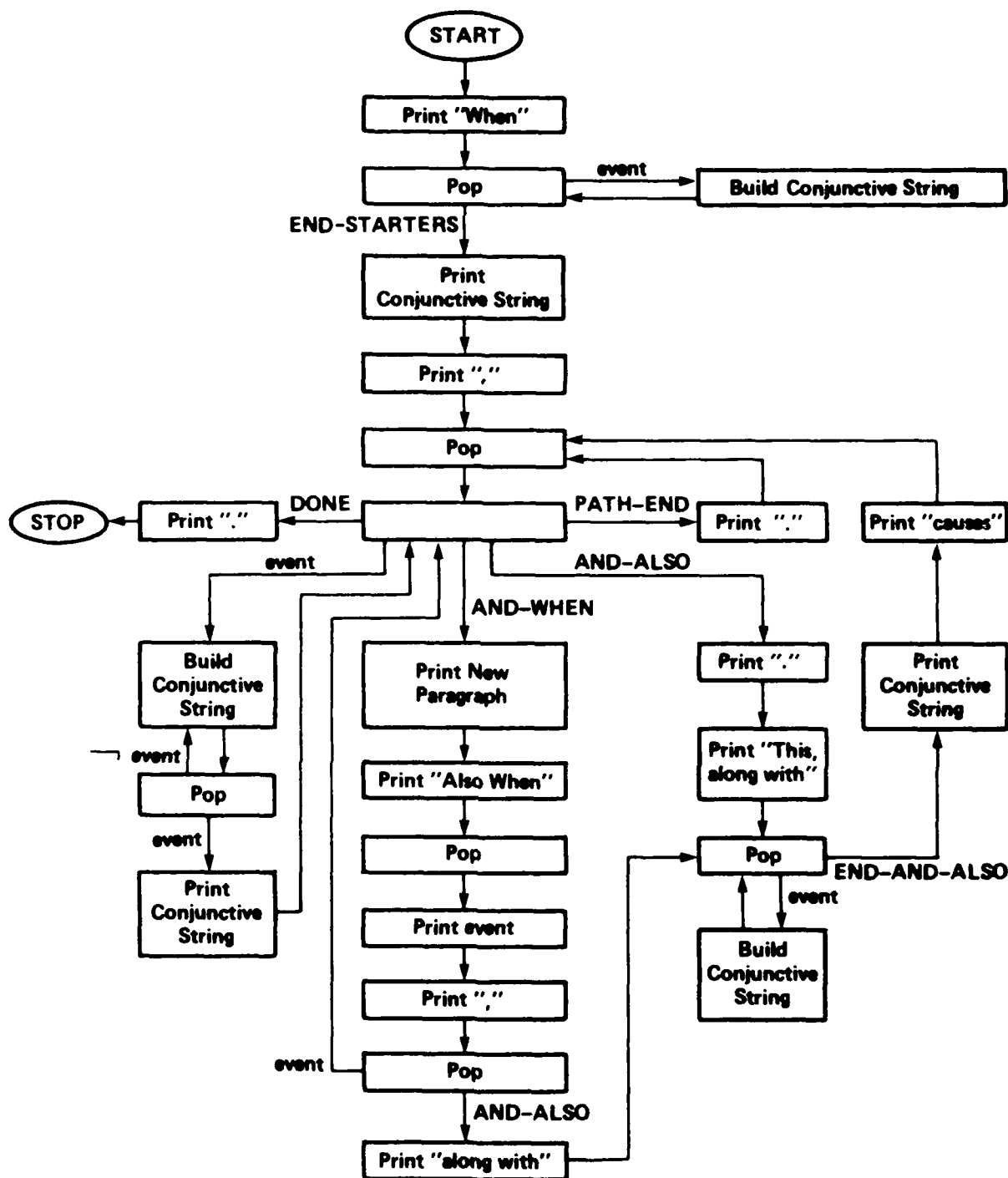
<sup>2</sup>It should be noted that clashes or coincidences may not directly refer to the controlled parameter. However, there must always be a path back to the parameter since it is the only one set to initiate the qualitative simulation. It is assumed that one of these paths would transmit the influence back to the parameter. This generally works well. However, it is possible that a device model may have a clash for another reason--it really might be open loop. A strategy that erases the alternate path and checks if the new value really does propagate back to the original one is intended for a future implementation.

of the network that represents it contains a U, D, or C. Events for which no string is returned are pruned.

Similarly, the English explanation is integrated with a graphical representation of the device by using a table of graphics functions. Each function is responsible for changing an appropriate part of the graphical representation of the device when it is called. In the spring reducer valve, for example, a graphics function associated with the pressure in the Piston Steam Port turns the contents either red, cyan, or white, depending on whether the pressure is increasing, decreasing, or constant. Each event that has a graphical representation causes a special graphics command string to be added to the string of phrases being constructed as the explanation. This interleaving of English and graphical presentation adds a great deal of clarity to the explanation.

The final step is to turn the collection of phrases into readable sentences and paragraphs. The string of phrases is parsed by a simple grammar that adds appropriate punctuation and breaks them up into paragraphs. The grammar used in the translation process is illustrated in Figure 3.

Results of analyzing the simulation are handled in the same fashion. A stored template provides an English explanation of the results, filled in with the phrases that describe the particular events in the device under consideration that led to the conclusions. For the feedback cases, events not on paths leading to the clash or coincidence are pruned from the explanation and a paragraph explaining the clash or coincidence is added. For the other cases, a standard paragraph is appended as part of the explanation stating whether the system is stable, unstable, or open-looped.



Pop takes the next element of the event string

Special markers in the string are indicated by capital letters

AND-WHEN refers back to a fork in the computation graph

AND-ALSO refers to a join in the computation graph

A conjunctive string has appropriate commas, such as "1",  
"1 and 2", "1, 2 and 3", etc.

Figure 3. The grammar that adds punctuation and turns phrases into sentences and paragraphs.

## CONCLUSIONS

It is possible to generate a coherent, understandable explanation of the operation of a physical device from a qualitative simulation of that device's operation, as the example documented in this report demonstrates. It is important to note that two tables (text and graphics) and the device description itself are the only device-dependent portions of this process. The qualitative simulation and its subsequent analysis are very general. New devices can easily be added by specifying their component connectivity and the text and graphics functions for each part.

The mixture of text and graphics animation provides a clarity not achieved by either in isolation. By interleaving the two, the system gains a great deal of power in generating an understandable explanation. The graphics animation serves to focus attention on the change being described. Its presentation within the context of a complete diagram unambiguously shows how the component responsible for the change relates to other components and previous changes.

The most important point to be made is that these techniques make possible learning environments in which students can experiment with complex devices and see explanations of the effects of various changes. Such learning environments would provide many features not available in normal instructional settings. Using a graphics interface, the student could manipulate parameters that would not be accessible in an actual device. (For example, the student could open the internal auxiliary valve in the reducing valve.) The qualitative nature of the simulation in effect provides the student with a personal tutor, interpreting for him the effects of the changes he makes, explaining and illustrating what happens inside the device.

Such a system could also be incorporated with other computer-based training systems. The possibility of using the techniques described in this report to aid procedures training is currently being examined. When practicing operational procedures, students make mistakes. A system of the type described here has the potential for providing

explanations of why some actions are correct, thereby causing the device to assume a desired state, and why others are incorrect or even dangerous, causing the device to assume undesired states.

One could even imagine constructing a "design laboratory" that enabled students to design and experiment with a device by putting together components. Students would not only be shown the operation of their device but also provided with an explanation of its operation. This level of "hands on" experience with internal and external device parameters could enable students to quickly understand complex physical systems in ways currently possible only after laborious study.

## REFERENCES

- Bureau of Naval Personnel. Principles of naval engineering. Washington, DC: U.S. Government Printing Office, 1970.
- deKleer, J. Causal and teleological reasoning in circuit recognition (PhD dissertation). Cambridge, MA: Massachusetts Institute of Technology, September, 1979. (b)
- deKleer, J. The origin and resolution of ambiguities in causal arguments. International Joint Conference on Artificial Intelligence, August 1979.
- Forbus, K. A study of qualitative and geometric knowledge in reasoning about motion (Master's thesis). Cambridge, MA: Massachusetts Institute of Technology, February 1980.
- Larkin, J., McDermott, J., Simon, D., & Simon, H. Expert and novice performance in solving physics problems. Science, June 1980, 208, 1335-1342.
- Stead, L. Project STEAMER: II. User's manual for the STEAMER interactive graphics package (NPRDC TN 81-22). San Diego: Navy Personnel Research and Development Center, August 1981.
- Steele, G., & Sussman, G. Constraints (AI Memo 502). Cambridge, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory, November 1978.
- Stevens, A., & Steinberg, C. Project STEAMER: I. Taxonomy for generating explanations of how to operate complex physical devices (NPRDC TN 81-21). San Diego: Navy Personnel Research and Development Center, August 1981.
- Stevens, A., Collins, A., & Goldin, S. Misconceptions in students' understanding. International Journal of Man-Machine Studies, 1979, 11, 145-156.

## **APPENDIX**

### **CONLAN: THE CONSTRAINT INTERPRETER**

## CONLAN: THE CONSTRAINT INTERPRETER

The qualitative simulation is implemented using CONLAN, a constraint interpreter that was originally developed by Guy Steele and Gerald Sussman at the Massachusetts Institute of Technology (MIT) and that has been extensively rewritten by Kenneth Forbus at Bolt Beranek and Newman, Inc. (BBN). It is written in Maclisp and is currently available at MIT and BBN.

### Conceptual View of CONLAN

Objects and the relationships between them are described in CONLAN by computational entities called constraints. A constraint has parts, some of which are other constraints and some of which are cells that hold values for the parameters in the description. An adder, for example, might have three cells--one for the sum and two for the addends. The computations that enforce the relationships implied by the use of the constraint are specified as rules that compute the value of a cell when some other cells are given values. The adder above would have three rules attached to it, because, given the values for any two of the cells, the value for the other cell can be computed.

Prototype constraints can be defined to facilitate building other constraints. Any constraint that is defined as a prototype can be instantiated into a network being constructed to describe some complex situation or used in turn as part of the definition of a more complex prototype.

Constraints are connected together to form descriptions by specifying that certain of their parts are actually the same. For example, an adder with three addends could be built from two of the adders described above by specifying that the sum of one of them is the same as one of the addends of the other. If defined as a constraint prototype, the three-argument adder could then be used as a component in still more complex constraints.

Computation in a constraint network occurs whenever a cell receives a value. The set of rules that use that cell is queued. Each rule in the set is then checked to see if it

can be run by checking to see if the cells it requires are known. If they are, the rule is run and the value, if not special, is placed in the cell the rule sets. Two special values are used--DISMISS, which indicates that the rule cannot yield a value, and LOSE, which indicates an inconsistent state of affairs in the network.

If there is already a value in the cell, the new value is checked against the old by a simple matching function (which can be user-defined for different cells if required). If the special value LOSE was returned or the values do not match, a clash is signalled. The default handler presents the premises underlying the clash to the user and asks which should be retracted. The dependency analysis is made possible by noting the source of the value whenever a cell is set. If a value is forgotten for any reason, all consequences of it are also removed and other ways of getting a value for that cell are investigated. This occurs by looking back from the rules that can set the cell for some rule that can be run.

#### Specifying Constraints

The syntax for a constraint prototype contains the name of the constraint, a list of the parts in it, and several pieces of information concerning the semantics of the parts in the particular description and their interconnections. An example of a prototype for an adder is illustrated in Figure A-1.

The key word "CONSTRAINT" indicates that this is a prototype definition. The name follows, and then a list of the names of the parts and what they are. The FORMULAE specification defines the computations that are particular to the parameters of the constraint. Each element consists of the cell that is to be set with the value, a list of the cells it requires to perform its computation, and the lisp code to evaluate once these cells are known. A rule that is to be executed purely for effect is indicated by a set-cell of NIL.

A uniform convention is used for referring to parts of a structure. The form (>> SUM ADD1) is interpreted to mean "the SUM OF ADD1." To specify that two parts are considered the same, the form (== <ref1> <ref2>) is used between any two parts of the



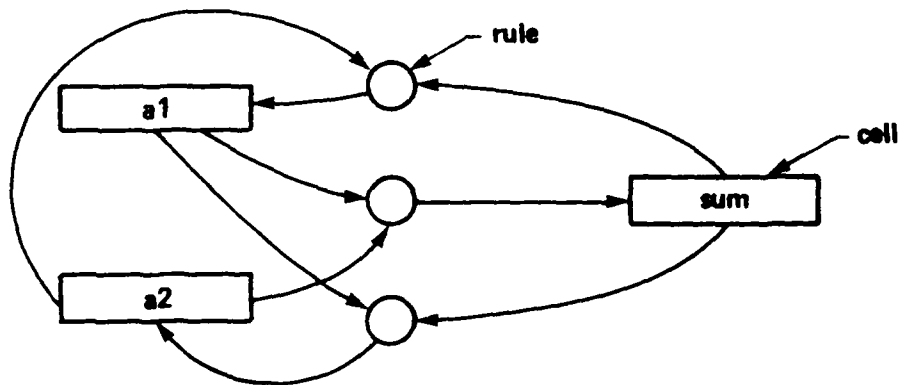
```

(constraint adder ((a1 cell)
                  (a2 cell)
                  (sum cell))

(formulae
  (sum (a1 a2) (+ a1 a2))
  (a1 (sum a2) (- sum a2))
  (a2 (sum a1) (- sum a1))))

```

Constraint prototype definition



Schematic of instantiated adder constraint

Figure A-1. Description of an adder.

same type. Statements of this type may be unbroken by (UN==<ref1> <ref2>). When defining a prototype, the form (R== <ref1> <ref2>) is used to say that those two names refer to exactly the same thing. During instantiation, only one object will be created but it will have both names.

The construction of a complicated network can be made easier by incorporating information about how constraints link together into the constraints themselves. Certain cells are considered as "indirects" because their values are other constraints. A special class of rules, called wiring rules, use the equality mechanism outlined above to make and

break appropriate connections between their parts when a value for an indirect cell is added or removed. The description of a pipe, for example, contains two indirect cells that are filled with objects that represent the things the pipe connects.

The Lisp program listing is provided on the following pages.

The Lisp Program Listing

;--LISP--

;;;Various string hacks.

```

(defun punctuation? (str)
  (cond ((or (string-equal str ".")
             (string-equal str ",")
             (string-equal str "!")
             (string-equal str ";")
             (string-equal str ":")) t)
        (t nil)))

(defun newline? (str)
  (string-equal str "
"))

(defun nthstring (n str)
  (substring str n (1+ n)))

(defun wordify (str)
  (do ((strprt 0 (1+ strprt))
      (len (string-length str))
      (word? nil)
      (words nil)
      (st ""))
      ((= strprt len)
       (setq words
              (cond (word? (nreverse (cons (apply 'string-append
                                                    (nreverse word?))
                                                    words))))
                (t (nreverse words))))
      (cond ((null words) (ncons str))
            (t words)))
    (setq st (nthstring strprt str))
    (cond ((punctuation? st)
           :pack it off as sepearte word
           (and word? (setq words
                             (cons (apply 'string-append
                                           (nreverse word?))
                                   words)))
           (push st words)
           (setq word? nil))
          ((string-equal " " st)
           (and word? (setq words
                             (cons (apply 'string-append
                                           (nreverse word?))
                                   words)))
           (setq word? nil))
          (t (setq word? (cons st word?))))))

(defun capitalize? (string flag)
  (cond (flag
        (do ((i 0 (1+ i))
            (len (string-length string))
            (done? nil)
            (substring ""))
            ((or done? (= i len)) string)
            (setq substring (substring string i (1+ i)))
            (cond ((string-equal " " substring)
                   (t (setq substring (string-upcase substring))
                     (string-replace string substring 1)
                     (setq done? t))))
            (t string)))
        (t string)))

```

```

(defun pretty-print-list-of-strings (strings)
  (let ((line-length (- (line1 nil) 3.)))
    (do ((str (cdr strings) (cdr str))
        (this (wordify (car strings)) (wordify (car str)))
        (line-left line-length)
        (stringpos 0)
        (stringlen 0)
        (last-word ""))
      ((null str) (terpri))
      (do ((words this (cdr words))
          (word "")
          (len 0))
          ((null words))
          ;first take action on the last word
          (setq word (car words))
          (len (string-length word))
          (cond ((string-equal word "&&&ESCAPE&&&")
                 ;don't print-this is a signal!
                 (and (boundp '*Print-escape-code*)
                      (funcall *print-escape-code*)))
                (t
                 (cond ((string-equal "." last-word) (princ " ")
                       (setq line-left (1- line-left)))
                       ((string-equal "." last-word) (princ " ")
                       (setq line-left (- line-left 2)))
                       ((not (or (string-equal word ".")
                                (string-equal word ".")
                                (string-equal word ";")
                                (newline? last-word)))
                        (princ " ")
                        (setq line-left (1- line-left))))
                 (cond ((newline? last-word)
                        (terpri)
                        (setq line-left line-length)))
                 (cond ((and (not (punctuation? word))
                             (> len line-left))
                        (terpri)
                        (setq line-left line-length)))
                 (cond ((or (and (string-equal last-word ".")
                                (string-equal word "."))
                            (and (string-equal word " "
                                (= line-left line-length))))
                        (t (princ (capitalize? word
                                (string-equal last-word "."))
                                (setq line-left (- line-left len))
                                (setq last-word word))))))))))
  ")

```

```
;-*-LISP*-;
```

```
;;;Specifications for Spring Reducer Valve used in  
;;;Feedback demo.
```

```
;;;Parts List for Graphics
```

```
(setq aeddraw:scale 24.0  
      background-color 4  
      free-space-color 7  
      valve-color 5  
      screen-color 0  
      spring-color 2  
      diaphram-color 3)
```

```
;screen offsets
```

```
(setq SCREEN:X-OFFSET 0.25  
      SCREEN:Y-OFFSET 0.5  
      SCREEN:SCALE 0.5)
```

```
(setq srv:parts
```

```
  ((Body (draw (Box 0 0 20 6)  
               (Box 2 6 16 16)  
               (Box 4 16 14 20)  
               (Box 4 20 9 22)  
               (Box 14 6 20 7))  
        (color . background-color))  
   (OutputPort (draw (Box 19 1 20 6))  
               (color . free-space-color))  
   (Chamber0 (draw (Box 12 1 19 6))  
             (color . free-space-color))  
   (MainValveOutput (draw (Box 9 4 12 6)  
                         (Box 7 4 8 6))  
                   (color . free-space-color))  
   (MainValveInput (draw (Box 6 1 11 3)  
                       (Triangle 2 7 3 8 4)  
                       (Triangle 3 9 4 10 3))  
                  (color . free-space-color))  
   (MainValve (draw (Box 6 9 11 10)  
                  (Box 8 2 9 9)  
                  (Triangle 2 7 2 8 3)  
                  (Triangle 3 9 3 10 2))  
             (color . valve-color))  
   (MainValveSpring (draw (Box 8.1 1.9 8.9 2.0)  
                        (Box 8.1 1.0 8.9 1.1)  
                        (Angle-Box-middle 8.5 1.5 -18.0 0.8 0.1))  
                   (color . spring-color))  
   (BelowPiston (draw (Box 6 7 8 9)  
                     (Box 9 7 11 9))  
               (color . free-space-color))  
   (Chamber4 (draw (Box 6 10 11 11))  
            (color . free-space-color))  
   (PistonSteamPort (draw (Box 9 11 10 16)  
                        (Box 8 15 9 16))  
                   (color . free-space-color))  
   (Chamber5 (draw (Box 5 18 6 19)  
                  (Box 7 18 8 19))  
            (color . free-space-color))  
   (AuxValveInput (draw (Box 5 12 8 13)  
                      (Box 5 13 8 14)  
                      (Triangle 2 5 14 6 15)  
                      (Triangle 3 7 15 8 14))  
                 (color . free-space-color))  
   (AuxValveOutput (draw (Box 5 15 6 17)  
                       (Box 7 15 8 17))  
                  (color . free-space-color))  
   (AuxValve (draw (Box 6 14 7 19)  
                 (Triangle 2 5.5 14 6 14.5)  
                 (Triangle 3 7 14.5 7.5 14))
```

```
(color . valve-color))
(AboveDiaphragm (draw (BOX 5 19.1 8 21))
  (color . free-space-color))
(AuxValveSpring (DRAW (BOX 6.1 20.9 6.9 21)
  (BOX 6.1 19.1 6.9 19.2):bot
  (ANGLE-BOX-middle 6.5 20 -30.0 0.8 0.1))
  (color . spring-color))
(Diaphragm (draw (BOX 5 19.0 8 19.1))
  (color . diaphragm-color))
(LowPressurePort (draw (Box 12 6 13 19)
  (Box 8 18 12 19))
  (color . free-space-color))
(HighPressurePort (draw (Box 3 5 4 14)
  (Box 4 13 5 14))
  (color . free-space-color))
(InputPort (draw (Box 0 1 5 5)
  (Box 5 1 7 3))
  (color . free-space-color))))
```

;Graphics Table

```
(setq U-color 1 :red
      D-color 6)
```

;when a cell is set, if the table entry exists, run the  
;graphics function

```
(setq *device-graphics-function-table*
      '(((Press Output-port) . (lambda (thing)
        (zap-part-color-and-blink 'OutputPort (value* thing))))
        ((Press Chamber0) . (lambda (thing)
        (zap-part-color-and-blink 'Chamber0 (value* thing))))
        ((Press2 Main-Valve) . (lambda (thing)
        (zap-part-color-and-blink 'MainValveOutput (value* thing))))
        ((press1 Main-Valve) . (lambda (thing)
        (zap-part-color-and-blink 'MainValveInput (value* thing))
        (aeddrow:draw-part 'MainValve)
        (aeddrow:draw-part 'MainValveSpring)))
        ((press chamber4) . (lambda (thing)
        (zap-part-color-and-blink 'Chamber4 (value* thing))))
        ((press Piston-Steam-Port) . (lambda (thing)
        (zap-part-color-and-blink 'PistonSteamPort (value* thing))))
        ((press Chamber5) . (lambda (thing)
        (zap-part-color-and-blink 'Chamber5 (value* thing))))
        ((press2 Aux-Valve) . (lambda (thing)
        (zap-part-color-and-blink 'AuxValveInput (value* thing))
        (aeddrow:draw-part 'AuxValve)))
        ((press1 Aux-Valve) . (lambda (thing)
        (zap-part-color-and-blink 'AuxValveOutput (value* thing))))
        ((delta-valve-state Aux-Valve) . (lambda (thing)
        (cond (*Using-AED*
              (caseq (value* thing)
                    (D (Move-Aux-Valve-Up))
                    (U (Move-Aux-Valve-Down))
                    ((C UNKNOWN) (Center-Aux-Valve)
                     nil)
                    (t nil))))))
        ((delta-valve-state Main-Valve) . (lambda (thing)
        (cond (*Using-AED*
              (caseq (value* thing)
                    (D (Move-Main-Valve-Up))
                    (U (Move-Main-Valve-Down))
                    ((C UNKNOWN) (Center-Main-Valve)
                     nil)
                    (t nil))))))
        ((press Low-Pressure-Port) . (lambda (thing)
        (zap-part-color-and-blink 'LowPressurePort (value* thing))))
        ((press High-Pressure-Port) . (lambda (thing)
        (zap-part-color-and-blink 'HighPressurePort (value* thing))))))
```

```
(defun zap-part-color (partname val)
  (cond (*Using-AED*
        (caseq val
              (U (aeddrow:change-part-color partname
                    U-color))
              (D (aeddrow:change-part-color partname
                    D-color))
              ((C UNKNOWN)
               (aeddrow:change-part-color partname
                    free-space-color))
              (t nil)))))
```

```
(defun zap-part-color-and-blink (partname val)
  (cond (*Using-AED*
        (caseq val
              (U (aeddrow:blink-and-change-part-color partname
```

```
                                U-color))
(D (aeddraw:blink-and-change-part-color partname
    D-color))
((C UNKNOWN)
 (aeddraw:blink-and-change-part-color partname
  free-space-color))
(t nil))))
```

```
(defun redraw-valve ()
(aeddraw:draw-parts srv:parts))
```

```
:::for different movements, make new things to draw
```

```
(defun generate-main-valve-shift (name amount)
(set name
'((Chamber4 (draw (Box 6 .(plus 10 amount) 11
11)))
(BelowPiston (draw (Box 6 .(plus 7 amount) 8 .(plus 9 amount))
(Box 9 .(plus 7 amount) 11 .(plus 9 amount)))))
(MainValveInput (draw (Box 6 1 11 3)
(Triangle 2 7 3 8 4)
(Triangle 3 9 4 10 3)))
(MainValveSpring (draw (Box 8.1 .(plus 1.9 amount) 8.9
.(plus 2.0 amount))
(Box 8.1 .(plus 1.0 amount) 8.9
.(plus 1.1 amount))
(Angle-Box 8.1 .(plus 1.75 amount)
.(plus (times 30.0 amount)
-30.0) 0.8 0.1)))
(MainValve (draw (Box 6 .(plus 9 amount) 11 .(plus 10 amount))
(Box 8 .(plus 2 amount) 9 .(plus 9 amount))
(Triangle 2 7 .(plus 2 amount) 8 .(plus 3 amount))
(Triangle 3 9 .(plus 3 amount) 10 .(plus 2 amount)))))))
```

```
(defun generate-aux-valve-shift (name amount)
(set name
'((Chamber5 (draw (Box 5 18 6 .(plus 19 amount))
(Box 7 18 8 .(plus 19 amount)))))
(AuxValveInput (draw (Box 5 12 8 14)
(Triangle 2 5 14 6 15)
(Triangle 3 7 15 8 14)))
(AuxValveOutput (draw (Box 5 15 6 17)
(Box 7 15 8 17)))
(AuxValve (draw (Box 6 .(plus 14 amount) 7 .(plus 19 amount))
(Triangle 2 5.5 .(plus 14 amount) 6
.(plus 14.5 amount))
(Triangle 3 7 .(plus 14.5 amount) 7.5
.(plus 14 amount)))))
(AboveDiaphragm (draw (Box 5 .(plus 19 amount) 8 21)))
(AuxValveSpring (draw (Box 6 20.8 7 21)
(Box 6 .(plus 19.2 amount) 7
.(plus 19.4 amount))
(Angle-Box 6 .(plus 20.6 amount)
.(plus (times amount 30.0)
-30.0) 1.0 0.2)))
(Diaphragm (draw (Box 5 18.8 8 19.2))))))
```

```
(defun Move-Main-Valve-Down ()  
  (mapc 'aeddrow:erase-part Main-Valve-Erase-List)  
  (aeddrow:draw-parts Main-Valve-Down-List))
```

```
(defun Center-Main-Valve ()  
  (mapc 'aeddrow:erase-part Main-Valve-Erase-List)  
  (aeddrow:draw-parts Main-Valve-Center-List))
```

```
(defun Move-Main-Valve-Up ()  
  (mapc 'aeddrow:erase-part Main-Valve-Erase-List)  
  (aeddrow:draw-parts Main-Valve-Up-List))
```

```
(defun Move-Aux-valve-Down ()  
  (mapc 'aeddrow:erase-part Aux-valve-Erase-List)  
  (aeddrow:draw-parts Aux-valve-Down-List))
```

```
(defun Center-Aux-valve ()  
  (mapc 'aeddrow:erase-part Aux-valve-Erase-List)  
  (aeddrow:draw-parts Aux-valve-Center-List))
```

```
(defun Move-Aux-valve-Up ()  
  (mapc 'aeddrow:erase-part Aux-valve-Erase-List)  
  (aeddrow:draw-parts Aux-valve-Up-List))
```

:::Coordinates for valve movement

(SETQ MAIN-VALVE-ERASE-LIST

'(MAINVALVE MAINVALVESPRING MAINVALVEINPUT BELOWPISTON CHAMBER4))

(SETQ MAIN-VALVE-UP-LIST

'((CHAMBER4 (DRAW (BOX 6 10.5 11 11)))  
 (BELOWPISTON (DRAW (BOX 6 7 8 9.5) (BOX 9 7 11 9.5)))  
 (MAINVALVEINPUT (DRAW (BOX 6 1 11 3)  
 (TRIANGLE 2 7 3 8 4)  
 (TRIANGLE 3 9 4 10 3)))  
 (MAINVALVESPRING (DRAW (BOX 8.1 2.4 8.9 2.5)  
 (BOX 8.1 1 8.9 1.1)  
 (ANGLE-BOX-MIDDLE 8.5 1.75 -60.0 0.8 0.1)))  
 (MAINVALVE (DRAW (BOX 6 9.5 11 10.5)  
 (BOX 8 2.5 9 9.5)  
 (TRIANGLE 2 7 2.5 8 3.5)  
 (TRIANGLE 3 9 3.5 10 2.5))))))

(SETQ MAIN-VALVE-CENTER-LIST

'((CHAMBER4 (DRAW (BOX 6 10.0 11 11)))  
 (BELOWPISTON (DRAW (BOX 6 7.0 8 9.0) (BOX 9 7.0 11 9.0)))  
 (MAINVALVEINPUT (DRAW (BOX 6 1 11 3)  
 (TRIANGLE 2 7 3 8 4)  
 (TRIANGLE 3 9 4 10 3)))  
 (MAINVALVESPRING (DRAW (BOX 8.1 1.9 8.9 2.0)  
 (BOX 8.1 1.0 8.9 1.1)  
 (Angle-Box-middle 8.5 1.5 -18.0 0.8 0.1)))  
 (MAINVALVE (DRAW (BOX 6 9.0 11 10.0)  
 (BOX 8 2.0 9 9.0)  
 (TRIANGLE 2 7 2.0 8 3.0)  
 (TRIANGLE 3 9 3.0 10 2.0))))))

(SETQ MAIN-VALVE-DOWN-LIST

'((CHAMBER4 (DRAW (BOX 6 9.5 11 11)))  
 (BELOWPISTON (DRAW (BOX 6 7.0 8 8.5) (BOX 9 7.0 11 8.5)))  
 (MAINVALVEINPUT (DRAW (BOX 6 1 11 3)  
 (TRIANGLE 2 7 3 8 4)  
 (TRIANGLE 3 9 4 10 3)))  
 (MAINVALVESPRING (DRAW (BOX 8.1 1.4 8.9 1.5)  
 (BOX 8.1 1.0 8.9 1.1)  
 (BOX 8.1 1.6 8.9 1.7)))  
 (MAINVALVE (DRAW (BOX 6 8.5 11 9.5)  
 (BOX 8 1.5 9 8.5)  
 (TRIANGLE 2 7 1.5 8 2.5)  
 (TRIANGLE 3 9 2.5 10 1.5))))))

(SETQ AUX-VALVE-ERASE-LIST

'(DIAPHRAM AUXVALVESPRING  
 ABOVEIDIAPHRAM  
 AUXVALVE  
 AUXVALVEOUTPUT  
 AUXVALVEINPUT  
 CHAMBER5))

:shift up by 0.5

(SETQ AUX-VALVE-UP-LIST

'((CHAMBER5 (DRAW (BOX 5 18 6 19.5) (BOX 7 18 8 19.5)))  
 (AUXVALVEINPUT (DRAW (BOX 5 12 8 14)  
 (BOX 6 14 7 14.5)  
 (TRIANGLE 2 5 14 6 15)  
 (TRIANGLE 3 7 15 8 14)))  
 (AUXVALVEOUTPUT (DRAW (BOX 5 15 6 17) (BOX 7 15 8 17)))  
 (AUXVALVE (DRAW (BOX 6 14.5 7 19.5)  
 (TRIANGLE 2 5.5 14.5 6 15.0)  
 (TRIANGLE 3 7 15.0 7.5 14.5)))  
 (ABOVEIDIAPHRAM (DRAW (BOX 5 19.6 8 21)))

```
(AUXVALVESPRING (DRAW (BOX 6.1 20.9 6.9 21)
                      (BOX 6.1 19.6 6.9 19.8)
                      (BOX 6.1 20.4 6.9 20.5)))
(DIAPHRAM (DRAW (BOX 5 19.5 8 19.6))))
```

## (SETQ AUX-VALVE-CENTER-LIST

```
((CHAMBERS (DRAW (BOX 5 18 6 19.0) (BOX 7 18 8 19.0)))
 (AUXVALVEINPUT (DRAW (BOX 5 12 8 14)
                     (TRIANGLE 2 5 14 6 15)
                     (TRIANGLE 3 7 15 8 14)))
 (AUXVALVEOUTPUT (DRAW (BOX 5 15 6 17) (BOX 7 15 8 17)))
 (AUXVALVE (DRAW (BOX 6 14.0 7 19.0)
                (TRIANGLE 2 5.5 14.0 6 14.5)
                (TRIANGLE 3 7 14.5 7.5 14.0)))
 (ABOVEDIAPHRAM (DRAW (BOX 5 19.1 8 21)))
 (AUXVALVESPRING (DRAW (BOX 6.1 20.9 6.9 21)
                      (BOX 6.1 19.1 6.9 19.2);bot
                      (ANGLE-BOX-middle 6.5 20 -30.0 0.8 0.1)))
 (DIAPHRAM (DRAW (BOX 5 19.0 8 19.1))))
```

## (SETQ AUX-VALVE-DOWN-LIST

```
((CHAMBERS (DRAW (BOX 5 18 6 18.5) (BOX 7 18 8 18.5)))
 (AUXVALVEINPUT (DRAW (BOX 5 12 8 14)
                     (TRIANGLE 2 5 14 6 15)
                     (TRIANGLE 3 7 15 8 14)))
 (AUXVALVEOUTPUT (DRAW (BOX 5 15 6 17) (BOX 7 15 8 17)))
 (AUXVALVE (DRAW (BOX 6 13.5 7 18.5)
                (TRIANGLE 2 5.5 13.5 6 14.0)
                (TRIANGLE 3 7 14.0 7.5 13.5)))
 (ABOVEDIAPHRAM (DRAW (BOX 5 18.6 8 21)))
 (AUXVALVESPRING (DRAW (BOX 6.1 20.9 6.9 21)
                      (BOX 6.1 18.6 6.9 18.7)
                      (ANGLE-BOX-middle 6.5 19.8 -60.0 0.8 0.1)))
 (DIAPHRAM (DRAW (BOX 5 18.5 8 18.6))))
```

## (setq Main-Valve-Invert-Center

```
((mainValveOutput (Draw (Box 9 4 12 6)
                       (Box 7 4 8 6)
                       (Triangle 4 7 4 8 3)
                       (Triangle 1 9 3 10 4)))
 (MainValve (Draw (Box 6 9 11 10)
                 (Box 8 2 9 9)
                 (Triangle 4 7 5 8 4)
                 (Triangle 1 9 4 10 5)))
 (MainValveInput (Draw (Box 6 1 11 3)))))
```

## (setq Main-Valve-Invert-Up

```
((mainValveOutput (Draw (Box 9 4 12 6)
                       (Box 7 4 8 6)
                       (Triangle 4 7 4 8 3)
                       (Triangle 1 9 3 10 4)))
 (MainValve (Draw (Box 6 9.5 11 10.5)
                 (Box 8 2.5 9 9.5)
                 (Triangle 4 7 5.5 8 4.5)
                 (Triangle 1 9 4.5 10 5.5)))
 (MainValveInput (Draw (Box 6 1 11 3)))))
```

## (setq Main-Valve-Invert-Down

```
((mainValveOutput (Draw (Box 9 4 12 6)
                       (Box 7 4 8 6)
                       (Triangle 4 7 4 8 3)
                       (Triangle 1 9 3 10 4)))
 (MainValve (Draw (Box 6 8.5 11 9.5)
                 (Box 8 1.2 9 8.5)
                 (Triangle 4 7 4.5 8 3.5)
                 (Triangle 1 9 3.5 10 4.5)))
 (MainValveInput (Draw (Box 6 1 11 3)))))
```

```
(setq Aux-Valve-Invert-Center
  '((AuxValveInput (draw (Box 5 12 8 13)
                        (Box 5 13 8 14)))
    (AuxValveOutput (draw (Box 5 15 6 17)
                        (Triangle 4 5 15 6 14)
                        (Triangle 1 7 14 8 15)))
    (AuxValve (Draw (Box 6 14 7 19)
                  (triangle 4 5.5 14.5 6 14)
                  (triangle 1 7 14 7.5 14.5))))))

(setq Aux-Valve-Invert-Up
  '((AuxValveInput (draw (Box 5 12 8 13)
                        (Box 5 13 8 14)))
    (AuxValveOutput (draw (Box 5 15 6 17)
                        (Triangle 4 5 15 6 14)
                        (Triangle 1 7 14 8 15)))
    (AuxValve (Draw (Box 6 14 7 19)
                  (triangle 4 5.5 14.5 6 14)
                  (triangle 1 7 14 7.5 14.5))))))

(setq Aux-Valve-Invert-Down
  '((AuxValveInput (draw (Box 5 12 8 13)
                        (Box 5 13 8 14)))
    (AuxValveOutput (draw (Box 5 15 6 17)
                        (Triangle 4 5 15 6 14)
                        (Triangle 1 7 14 8 15)))
    (AuxValve (Draw (Box 6 14 7 19)
                  (triangle 4 5.5 14.5 6 14)
                  (triangle 1 7 14 7.5 14.5))))))
```

;-\*-lisp\*-;

;;Using IQ rules to demonstrate feedback.  
;;Additional state information is required to  
;;describe the context of the encounter.  
;;The pieces are-  
;;  
;;\*DEVICE\*, which is the device under consideration.  
;;Its properties are singled out for incremental change  
;;to discover if and how the system restores itself.  
;;  
;;\*FOCUS\* is the quantity being diddled.  
;;This code traces paths back from a clash to this  
;;value. The path is comprised of the rules which  
;;lead from the FOCUS value to the value under  
;;consideration.

```
(declare (*lexpr say-it))  
(includef '((forbus) lmacro lsp))  
(includef '((lstead) aed dcls))  
(includef '((lstead) gl dcls))  
(includef '((lstead) turtle dcls))
```

```

:::Top level loop for feedback trainer
::
:::commands are:
:::CONSIDER DEVICE <prototype>
:: Which sets *DEVICE* up, as well as the printing tables for it
::
:::CONSIDER <quantity>
:: Which must be an IQ cell within *DEVICE*. This will become
:: the focus of the dialog, and is set to *FOCUS*
::
:::SUPPOSE <U,D,?,C>
:: Which sets the *FOCUS* to the value and triggers the analysis.
:: The constraint interpreter is fired, and the results summarized.
:: The trick will be to leave a very large amount of state information
:: around for other commands.
::
:::EXPLAIN
:: Gives a canned explanation of whatever had happened as a result
:: of the analysis.
::
::
;insure domestic tranquility
(setq *TTY-ECHO* T
      *USING-AED* nil
      *AED-ECHO* nil)

(defun feedback-trainer ()
  (setq *conlan-contradiction-handler* 'Feedback-clash-handler)
  (clear-text-window)
  (new-aed-line)
  (princ-aed "->")
  (do ((form (read) (read)))
      ((memq form '(quit stop)) T)
      (cond ((symbolp form)
              (caseq form
                ((INIT INITIALIZE BEGIN START)
                 (echo-aed form)
                 ;clear everything
                 (conlan-init)
                 (configure-graphics)
                 (cond (*using-AED*
                       ;these are defined elsewhere
                       (clear-text-window)
                       (clear-graphics-window)))
                 (clear-feedback-analysis))
                (CONSIDER
                 ;echo input on AED if necessary
                 (echo-aed form)(echo-aed " ")
                 ;either a new part or a new device
                 (let ((thing (read))
                       (new nil)
                       (name nil))
                   (echo-aed thing)(echo-aed " ")
                   (cond ((memq thing '(device component part))
                         (setq new (read))
                         (echo-aed new)(echo-aed " ")
                         (cond ((ok-prototype? new)
                               (new-aed-line)
                               (princ-aed "What should the ")
                               (princ-aed (to-string new))
                               (princ-aed " be called?")
                               (new-aed-line)
                               (setq name (read))
                               (princ-aed name)
                               (new-aed-line)
                               (cond ((boundp '*device*)
                                     (destroy-constraint *device*))))
                               (t)
                               (t))))
                         (t)
                         (t))))
                   (t)
                   (t))))
                 (t)
                 (t))))
              (t)
              (t))))

```

```

        (princ-aed
          (setq *device*
            (eval '(create .name '.new))))
        (fire-constraints) ;wire it up
        (setq *DEVICE-STATUS-ALIST* NIL)
        (clear-feedback-analysis))
      (t (say Sorry - $ new unknown component))))
    (t ;must be an IQ cell
      (cond ((not (boundp '*device*))
        (say No device has been specified))
        ((error? (list 'eval thing))
          (say Sorry - $ thing is
            incomprehensible))
        (t (setq new (eval
          (list 'eval thing)))
          (cond ((not (IQ-cell? new))
            (say Sorry - $ (p-name new) is
              not comprehended in this way))
            ((not (subconstraint? new *device*))
              (say Sorry - $ (p-name new) is not
                part of the device
                under consideration))
            (t ;actually a legal choice
              (clear-feedback-analysis)
              (setq *focus* new)
              (setq *connected-to-focus*
                (find-connected-set *focus*))
              ))))))
    (SUPPOSE ;takes an IQ value, sets the *FOCUS* to it
      ;and runs CONLAN over the network. State variables
      ;for explanation must be set up here.
      (echo-aed form)
      (clear-feedback-analysis)
      (let ((val (read)))
        (echo-aed val)
        (cond ((not (memq val '(U D ? C)))
          (new-aed-line)
          (princ-aed "Sorry - the value must be one")
          (princ-aed " of U, D, C, or ?.")
          (t (cond ((known? *focus*)
            (forget! *focus*
              (informant? *focus*))))
            (set! *focus* val 'FROB);ahem!
            (draw-device)
            (fire-constraints)
            (analyze-feedback-constraint-network)
            (print-feedback-analysis))))))
    (DESCRIBE
      (echo-AED form)
      (explain-the-events *event-string*
        *graphics-list*))
    (EXPLAIN
      (cond (*using-aed*
        (princ-AED "Explain")))
      (cond ((boundp '*feedback-event-string*
        (explain-the-events *feedback-event-string*
          *feedback-graphics-list*))
        (t (explain-the-events *event-string*
          *graphics-list*)
          (print-feedback-analysis))))
    ((? HELP)
      (cond (*using-aed*
        (princ-aed "Help")))
      (cond ((or (not (boundp '*using-AED*))
        (null *using-AED*))
        (terpri)
        (princ "The commands are:")
        (terpri)
        (princ "BEGIN, which clears the environment")

```

```

(terpri)
(princ "QUIT, which stops the trainer")
(terpri)
(princ "CONSIDER <device>, which asks for a name")
(terpri)
(princ "CONSIDER <quantity>, which must be a part of the device")
(terpri)
(princ "SUPPOSE <val>, which must be one of")
(terpri)
(princ "      U for increasing")
(terpri)
(princ "      D for decreasing")
(terpri)
(princ "      C for constant")
(terpri)
(princ "      ? for indeterminate")
(terpri)
(princ "EXPLAIN, which prints the argument used in ")
(terpri)
(princ "deciding how the quantity will behave.")
(terpri)
(princ "DESCRIBE, which prints a description of the ")
(terpri)
(princ "behaviour of the device.")
(terpri)
(princ "CLEAR, which initializes the display"))
(t ;use AED
  (new-AED-line)
  (princ-AED "The commands are:")
  (new-AED-line)
  (princ-AED "BEGIN, which clears the environment")
  (new-AED-line)
  (princ-AED "QUIT, which stops the trainer")
  (new-AED-line)
  (princ-AED "CONSIDER <device>, which asks for a name")
  (new-AED-line)
  (princ-AED "CONSIDER <quantity>, which must be a part of the device")
  (new-AED-line)
  (princ-AED "SUPPOSE <val>, which must be one of")
  (new-AED-line)
  (princ-AED "      U for increasing")
  (new-AED-line)
  (princ-AED "      D for decreasing")
  (new-AED-line)
  (princ-AED "      C for constant")
  (new-AED-line)
  (princ-AED "      ? for indeterminate")
  (new-AED-line)
  (princ-AED "EXPLAIN, which prints the argument used in ")
  (new-AED-line)
  (princ-AED "deciding how the quantity will behave.")
  (new-AED-line)
  (princ-AED "DESCRIBE, which prints a description of the ")
  (new-AED-line)
  (princ-AED "behaviour of the device.")
  (new-AED-line)
  (princ-AED "CLEAR, which initializes the display"))))
(CLEAR (cond (*using-aed*
              (initturtle))))
(redraw (cond (*using-aed*
              (redraw-device))))
(t      (new-aed-line)
        (princ-aed (eval form)))
)
(t      (new-aed-line)
        (princ-aed (eval form))))
(new-aed-line)
(princ-aed "->"))

```

```

:::Support functions for Feedback-Trainer driver loop
::
::Organized by command
::
::INIT

(defun configure-graphics ()
  (terpri)
  (princ "Use the AED?")
  (cond ((Read-yes-or-no?)
    (initturtle)
    ;must put some sort of load check in here
    (start-blinker)
    (setq *using-AED* t
          *incremental-graphics* t
          *AED-ECHO* t)
    (princ "Echo on TTY also?")
    (cond ((Read-yes-or-no?)
      (setq *TTY-ECHO* t)
      (t (setq *TTY-ECHO* nil))))
    (t (setq *using-AED* nil
          *incremental-graphics* nil
          *AED-ECHO* nil
          *TTY-ECHO* T))))

(defun read-yes-or-no? ()
  (let ((ans (read)))
    (cond ((nengq ans '(Y y yes yup yeah affirmative)) T)
          ((nengq ans '(N n no nope naw negative)) NIL)
          (t (Print "Please type yes or no")
             (read-yes-or-no?))))))

(defun clear-feedback-analysis ()
  (mapc 'makunbound '(*FEEDBACK-STATUS* *INFLUENCES*
                    *EVENTS* *STARTERS*
                    *CLEANED-EVENTS* *EVENT-STRING*
                    *FEEDBACK-EVENTS*
                    *FEEDBACK-CLEANED-EVENTS*
                    *FEEDBACK-EVENT-STRING*
                    *GRAPHICS-LIST*
                    *FEEDBACK-GRAPHICS-LIST*)))

```

```
;;;for CONSIDER
;;
```

```
(defun ok-prototype? (name)
  (fetch-prototype-name name))

(defun IQ-cell? (cell)
  (equal (get cell 'values) '(U D ? C)))

(defun subconstraint? (part? constraint)
  ;yes, this is gross
  (cond ((null part?) nil)
        ((eq part? constraint) t)
        (t (apply 'or
                   (mapcar '(lambda (x)
                             (subconstraint? part? x))
                           (mapcar '(lambda (partname)
                                     (get constraint partname))
                                   (partnames? constraint)))))))
```

```
;;;Finding connected set of a cell
;;In the IQ constraint models, equality is used to
;;indicate physical connection. (While servicable,
;;an explicit connection statement will be needed for
;;heat as well as fluid) Given a cell the cells connected
;;to it must be found to perform feedback analysis (for the
;;clash need not be in the focus cell).
```

```
(defun find-connected-set (cell)
  (do ((cells (ncons cell) (nconc newcells (cdr cells)))
      (connected nil)
      (quantity-type (get cell 'quantity-type))
      (newcells nil nil))
      ((null cells) connected)
      (push (car cells) connected)
      (mapc '(lambda (rule)
               (let ((cell (sets? rule)))
                 (cond ((and (memq (car rule) '(1<-2 2<-1));equality?
                              (equal (get cell 'quantity-type)
                                     quantity-type) ;same kind of thing?
                              (not (memq cell connected))
                              (not (memq cell newcells)));paranoid
                       (push cell newcells))))))
            (users? (car cells)))))
```

```
;;;SUPPOSE
;;
```

```
(defun draw-device ()
  (and (boundp '*using-AED*
        *using-AED*
        (aeddrow:draw-parts aeddrow:parts)))

(defun redraw-device ()
  (and (boundp '*using-AED*
        *using-AED*
        (m=pc 'aeddrow:draw-part aeddrow:parts)))
```

```
;;;Analyzing feedback with a constraint network
;;;description of IQ models.
;The FOCUS is given some new value, and the CONLAN
;interpreter is fired up. Either the network will
;reach quiescence or a clash will occur.
;IF A CLASH - if the clash is independent of the focus, something
;              is wrong with the other assumptions, and must be
;              corrected by the user before any feedback analysis
;              can be performed.
;              if the clash involves the focus, either the values are U and D
;              (in which case the system is exhibiting negative feedback)
;              or C and either D or U, in which case the system will be
;              unstable. If one of them is ?, the trainer is confused.
;IF NO CLASH - if there is another supplier for the focus which returns a value
;              consistent with it (and these values are either U or D),
;              there is positive feedback. If both are C the system is
;              stable in the steady state.
;              if there isn't another supplier for the focus, the system is
;              an open-loop system, without feedback as far as the
;              trainer's models are concerned.
```

```
;;;State Variables left around by analysis-
;;*FEEDBACK-STATUS* - either OPEN-LOOP, STABLE, UNSTABLE,
;;                  POSITIVE-FEEDBACK, NEGATIVE-FEEDBACK.
;;*INFLUENCES*      - A list of dotted pairs, whose CAR is the
;;                  cell and whose CDR is the rule that set the
;;                  cell. The interpretation of the influences
;;                  depends on the feedback-status.
;;*DEVICE-STATUS-ALIST* - an Alist of values and the type of feedback
;;                  behaviour the system exhibits with that value.
```

```
(defun analyze-feedback-constraint-network ()
  ;;by this time the constraint interpreter has already been fired
  ;;check it anyway
  (cond ((or (not (boundp '*device*))
             (not (boundp '*focus*))
             (not (known? *focus*)))
        (say Insufficient data for feedback analysis))
    (t (let ((initial-value (value? *focus*)))
        (find-loop-rule) ;look for places of feedback
        (cond ((eq *conlan-status* 'QUIESCENT)
              ;either OPEN-LOOP, POSITIVE-FEEDBACK, or STABLE.
              (cond ((boundp '*challenger*)
                     (destroy-constraint *challenger*)
                     (makunbound *challenger*))) ;keep state correct
                (let ((suppliers (accumulate
                                   '(lambda (rule)
                                       (eq (evaluate-rule rule)
                                             initial-value))
                                   *loop-rules*)))
                  (cond ((null suppliers) ;Open loop case
                        (setq *feedback-status* 'OPEN-LOOP)
                        )
                    )
                )
        )
```

```

      ((eq initial-value 'c)
       (setq *feedback-status* 'STABLE))
      ((eq initial-value '?)
       (setq *feedback-status* 'unknown))
      (t (setq *feedback-status*
               'POSITIVE-FEEDBACK)
         (setq *influences*
               (mapcar '(lambda (x)
                          (cons (sets? x) x))
                       suppliers))
          ;want path to include the place where
          ;feedback occurs
          (setq *challenger*
                (ccell initial-value
                      (car suppliers)
                      nil))))
      (push (cons initial-value
                  (*feedback-status*
                   *DEVICE-STATUS-ALIST*)))
      ((and (eq (typep *conlan-status*) 'list)
            (eq (car *conlan-status*) 'CLASH))
       (setq *challenger*
             (car (accumulate '(lambda (x)
                                (challenger? x))
                              (cadr *conlan-status*)))))
      (cond ((eq initial-value 'C)
             (setq *feedback-status* 'UNSTABLE))
            ((eq initial-value '?)
             (setq *feedback-status* 'OPEN-LOOP))
            ((memq initial-value '(U D))
             (setq *feedback-status* 'NEGATIVE-FEEDBACK))
            (t (setq *feedback-status* 'BUG-RIDDEN)))
      (setq *influences*
            (mapcar '(lambda (x)
                       (cons x (informant? x)))
                    (cadr *conlan-status*)))
      (push (cons initial-value
                  (*FEEDBACK-STATUS*
                   *DEVICE-STATUS-ALIST*)))
      (t (makunbound '*challenger*)
         (say Constraint network is in a wedged state
              - cannot analyze))))
      (setq *events* (forward-explanation *focus*))
      (setq *cleaned-events* (clean-up-event-list *events*))
      ;keep full explanation
      (transform-events-to-strings
       *cleaned-events* '*event-string* '*graphics-list*)
      (cond ((memq *feedback-status* '(POSITIVE-FEEDBACK
                                       NEGATIVE-FEEDBACK))
            ;make special feedback explanation
            (setq *feedback-events* (prune-non-loop-events *events*)
                  *feedback-cleaned-events* (clean-up-event-list
                                             *feedback-events*)))
            (t
             ;has to set two things
             (transform-events-to-strings
              *feedback-cleaned-events*
              '*feedback-event-string*
              '*feedback-graphics-list*)
             (setq *feedback-event-string*
                   (append *feedback-event-string*
                           (feedback-capper *feedback-status*
                                              *focus*
                                              *challenger*))))))

```

```

;;;A LOOP-RULE is a rule which comes back to the connected
;;set. If the value it produces is the same as the cell in
;;the connected set, it indicates positive feedback. If
;;not, it SHOULD BE the rule producing the challenger.
;;Here we just find the loop rule from the connected set

```

```

(defun find-loop-rule ()
  (cond ((or (not (boundp '*focus*))
             (not (boundp '*connected-to-focus*))))
        (t ;preconditions exist
         (let ((rules (scratch (apply 'append
                                       (mapcar 'suppliers? *connected-to-focus*))))))
           ;filter
           (setq rules (accumulate
                        '(lambda (rule)
                          (and (memq (sets? rule) *connected-to-focus*)
                               (accumulate '(lambda (cell)
                                              (not (memq cell *connected-to-focus*)))
                                              (uses? rule))
                               (not (memq (car rule) '(1<-2 2<-1)))
                               (runnable? rule)
                               (not (wiring-rule? rule))
                               (sets? rule) ;not for effect
                               (not (memq (evaluate-rule rule)
                                           '(*dismiss* *lose*))))))
                        rules))
           (setq *loop-rules* rules))))))

```

:::This prints a canned summary of the analysis

```
(defun print-feedback-analysis ()
  (cond ((or (not (boundp '*FEEDBACK-STATUS*))
             (not (boundp '*FOCUS*))
             (not (known? *FOCUS*)))
        (new-AED-line)
        (princ-AED "System not yet analyzed.))
    (t
     (clear-text-window)
     (caseq *FEEDBACK-STATUS*
      ((NEGATIVE-FEEDBACK
        POSITIVE-FEEDBACK)
       (Pretty-Print-to-AED
        (list "
          "
          "When the value is "
          (IQ-print-val-ing (value? *focus*))
          " " "the system tries to make it "
          (IQ-print-val (value? *challenger*))
          " "
          "This means it has " (to-string *FEEDBACK-STATUS*)
          " ")))
        (OPEN-LOOP
         (pretty-print-to-AED
          (list "
            "What happened did not affect the value" " "
            "so the system is open loop" " ")))
        (UNKNOWN (new-AED-line)
                  (pretty-print-to-AED
                   (list "
                     "What happened was independent of the value.)))
        (STABLE (pretty-print-to-AED
                  (list "
                    "Nothing in the system causes the value"
                    "to change when it is constant" " " "so it is stable" " ")))
        (BUG-RIDDEN (pretty-print-to-AED
                      (list "
                        "Things are very confused" " " "HELP!"))))))))
```

```
(defun opposite-IQ-val (val)
  (cdr (assoc val '((U . D)(D . U)(? . ?)(C . C)))))
```

```
(defun IQ-print-val-ing (val)
  (cdr (assoc val '((U . "increasing")(D . "decreasing")
                  (? . "indeterminate")
                  (C . "constant")))))
```

```
(defun IQ-print-val (val)
  (cdr (assoc val '((U . "increase")(D . "decrease")
                  (? . "indeterminate")
                  (C . "constant")))))
```

```

:::Forward "explanation generator"
::Starting from FOCUS, goes through the known values in
::depth first fashion. The result is a list of cell-rule pairs
::which constitute "what happened" in the network as a result
::of the stated change.

```

```

:EXACT FORMAT -

```

```

:*STARTERS* -- What the initial rule depended on
:*EVENTS* -- ((<cell> . <rule>)...) in order to be printed.

```

```

:first pass - ignore challenger

```

```

(defun forward-explanation (cell)
  (cond ((or (not (cellp? cell))
             (not (known? cell))
             (not (boundp '*device*))))
        (say $ cell is inexplicable to me))
    (t (clear-consequence-markers *device*)
        (mark-consequences cell)
        (setq *EVENTS* nil)
        (let ((users (accumulate 'runnable?
                                  (users? cell))))
          (cond ((null users)
                 (setq *starters* (ncons cell)
                       *events* nil))
                (t (setq *starters*
                          (accumulate '(lambda (x)
                                          (atom (informant? x)))
                                      (uses? (car users))))
                  (gather-forward-events (car users)
                                          (sets? (car users)))
                  *events*)))))))

```

:::Support functions for forward explanation generator

```

(defun clear-consequence-markers (constraint)
  (do ((parts (ncons constraint)
              (append (cdr parts) newparts))
      (newparts nil)
      (part nil))
      ((null parts) constraint)
      (setq part (car parts))
      (remprop part 'consequence-marker)
      (remprop part 'print-marker)
      (setq newparts (mapcar '(lambda (name) (get part name))
                             (partnames? part)))))

(defun mark-consequences (cell)
  (do ((cells (ncons cell) (append (cdr cells)
                                     newcells))
      (newcells nil)
      (cc nil))
      ((null cells) cell)
      (setq cc (car cells))
      (cond ((or (not (known? cc))
                 (get cc 'consequence-marker))) ;be lazy
            (t (putprop cc t 'consequence-marker)
               (setq newcells
                     (accumulate '(lambda (set)
                                   (and (not (atom (informant? set)))
                                       (memq cc
                                             (uses? (informant? set)))))
                                (mapcar '(lambda (rule) (sets? rule))
                                        (users? cc)))))))

(defun join? (rule) (> (length (accumulate '(lambda (cell)
                                              (get cell 'consequence-marker))
                                              (uses? rule)))
                        1.))

```

```

(defun gather-forward-events (rule cell)
  (do ((event-queue (list (cons cell rule))
      (append front (cdr event-queue) back))
      (c nil);current cell
      (r nil);current rule
      (skip-rest? nil nil)
      (front nil nil)
      (back nil nil)
      (others nil nil)
      (challenger-informant (cond ((boundp '*challenger*)
      (informant? *challenger*))))
      (possibles nil nil))
    ((null event-queue) (setq *events*
      (nreverse (cons '(DONE) *events*))))
    (setq c (caar event-queue)
      r (cdar event-queue))
    (cond ((null c) ;rule executed only for effect
      (push (cons 'EFFECT r) *events*))
      ((join? r) ;must see if others are around
      (setq others
        (accumulate '(lambda (x) (and (not (equal x c))
      (get x 'consequence-marker)
      (not (get x 'print-marker))))
      (uses? r)))
      (cond (others ;still must have other events occur first
        ;place the current on the back of the queue
        (setq back (ncons (car event-queue))
      skip-rest? t))
        (t (push (cons 'JOIN
      (accumulate
        '(lambda (x)
      (and (not (equal x c))
      (get x 'consequence-marker)))
      (uses? r)))
      *events*))))))
      ((or skip-rest?
      (get c 'print-marker))
      (t (push (cons c r) *events*)
      (put-top c t 'print-marker)
      (setq possibles (delete nil
      (mapcar '(lambda (x)
      (cond ((equal (informant? (sets? x)) x)
      (cons (sets? x) x))
      ((equal challenger-informant x)
      (cons *challenger* x))))
      (users? c))))))
      (cond ((null possibles);end of a path
      (push (ncons 'path-end) *events*))
      ((= (length possibles) 1)
      (setq front (ncons (car possibles))))
      (t :fork
      (push (cons 'FORK possibles) *events*)
      (setq front possibles))))))

```

```

:::Printing hack
::In producing explanations, a cell and its value must result
::in a string that can be printed. Attached to the device
::constraint must be a table of part names (or rule names)
::and an associated
::function that when run on the thing will produce a string
::to be printed.
::Other phrases will be introduced according to the connectivity
::of the dependency graph as glue.

```

```

(defun get-printing-function (name)
  (cond ((null name) nil)
        ((atom name) (break foo! t))
        ((atom (car name)) :simple name
         (let ((stuff (cdr (assoc name *device-print-function-table*))))
           (cond ((atom stuff) (symeval stuff))
                 (t stuff))))
        (t (let ((stuff (accumulate '(lambda (x) x)
                                      (mapcar '(lambda (simple)
                                                (cdr (assoc simple
                                                              *device-print-function-table*)))
                                              name))))
          (cond ((= (length stuff) 1)
                (cond ((atom (car stuff))
                      (symeval (car stuff)))
                  (t (car stuff))))
                (t nil)))))))

```

```

(defun simplify-name (name)
  ;;the front ">>" must be removed and the particular
  ;;name for the device constraint must be removed.
  (cond ((or (null name)
            (atom name)))
        ((atom (cadr name)) :simple name
         (cond ((eq (car name) '>>) :part
              (reverse (cdr (reverse (cdr name)))))
              ((eq (cadr name) '>>) :rule
              (cons (car name)
                    (reverse (cdr (reverse (cddr name))))))
              (t (break simplify-name t))))
        (t :compound name
         (mapcar '(lambda (x) (reverse (cdr (reverse x))))
                 (cdr name)))))

```

```

(defun feedback-clash-handler (disputants)
  (let ((assumptions (assumptions disputants)))
    (cond ((not (boundp *focus*))
          (user-break-contradiction-handler disputants))
          ((not (memq *focus* assumptions))
          (say There is a problem with the system)
          (user-break-contradiction-handler disputants))
          (t (setq *CONLAN-STATUS* (list 'CLASH
                                          disputants)
                  *STOP-CONLAN* t))))))

```

```

(defun feedback-capper (type focus challenger)
:produces snappy string at the end of the argument
(let ((graphics nil))
(setq *graphics-events-accumulator* nil)
(setq graphics
(cond ((eq type 'POSITIVE-FEEDBACK)
      (append '("
        "
        "Notice that when")
        (nreverse (make-event-string focus))
        '("
        "it means")
        (nreverse (make-event-string (sets? (informant? challenger))))
        '("
        "However, it also causes the system to act so that")
        (nreverse (make-event-string challenger))
        '("
        "This means it exhibits positive feedback" "."))))
      ((eq type 'NEGATIVE-FEEDBACK)
       (append '("
        "
        "Notice that when")
        (nreverse (make-event-string focus))
        '("
        "it means")
        (nreverse (make-event-string (sets? (informant? challenger))))
        '("
        "However, it also causes the system to act so that")
        (nreverse (make-event-string challenger))
        '("
        "This means it exhibits negative feedback" "."))))
      (T NIL)))
(setq *feedback-graphics-list*
      (nconc *feedback-graphics-list*
              (nreverse *graphics-events-accumulator*))))
graphics))

```

```

(defun convert-to-path ()
  (delete nil (mapcar '(lambda (x)
                        (caseq (car x)
                              ((path-end done) nil)
                              ((fork join) nil)
                              (t (cdr x))))
                    *events*)))

```

```

(defun princ-blanks (number)
  (do i 1 (1+ i) (> i number)
    (princ " ")))

```

```

(defun simple-print (pair)
  (let ((rp (get-printing-function (simplify-name (p-name (cdr pair)))))
        (cp (get-printing-function (simplify-name (p-name (car pair)))))
        (cond (rp (explain-print rp (cdr pair))))
        (cond (cp (explain-print cp (car pair)))))

```

```

(defun find-printing-function (thing)
  (cond ((atom thing) ;cell or challenger
        (cond ((challenger? thing)
              (get-printing-function
               (simplify-name (p-name (sets? (informant? thing)))))
              (t (get-printing-function (simplify-name (p-name thing)))))
        (t ;rule
          (get-printing-function (simplify-name (p-name thing)))))

```

```

(defun challenger? (thing)
  (and (atom thing) (eq (car (get thing 'name)) 'challenger)))

```

```

(defun find-graphics-function (thing)
  (let ((gfun nil))
    (setq gfun
      (cond ((atom thing) ;cell or challenger
        (cond ((challenger? thing)
          (get-graphics-function
            (simplify-name (p-name (sets? (informant? thing))))))
          (t (get-graphics-function (simplify-name (p-name thing)))))
        (t ;rule
          (get-graphics-function (simplify-name (p-name thing)))))
      (cond (gfun (list gfun thing))
        (t nil))))

```

```

(defun get-graphics-function (name)
  (cond ((null name) nil)
    ((atom name) (break fool t))
    ((atom (car name)) ;simple name
      (let ((stuff (cdr (assoc name *device-graphics-function-table*))))
        (cond ((atom stuff) (syneval stuff))
          (t stuff))))
    (t (let ((stuff (accumulate '(lambda (x) x)
      (mapcar '(lambda (simple)
        (cdr (assoc simple
          *device-graphics-function-table*)))
        name))))
      (cond ((= (length stuff) 1)
        (cond ((atom (car stuff))
          (syneval (car stuff)))
          (t (car stuff))))
        (t nil))))))

```

:::Printing trace of events

```
(defun lookup-fork (key forks)
:format of fork list is (<starters> . <pairs>)
(do ((ff forks (cdr ff))
      (result nil))
    ((or result (null ff)) result)
    (cond ((member key (cdar ff)) (setq result (caar ff))))))

(defun lookup-join (key joins)
(cdr (assoc key joins)))
```

```

:::Turning event list into prettier English
::First step is to prune out those parts which will
::have no printed representation, as evidenced by the
::print function table.
::Then the extra parts necessitated by the fork and
::join statements are placed in the stream.
::Trimming may be required, and then punctuation is
::added.
::*CLEANED-EVENTS* is produced by this process
::
::*CONNECTIONS-TO-FOCUS* are the set of equality rules
::whose purpose is to transmit influences to and from
::the focus. It is computed by a separate process when
::the focus is chosen.

```

```

(defun clean-up-event-list (*events*)
  (cond ((or (not (boundp '*focus*))
             (not (boundp '*events*))
             (not (boundp '*starters*))))
        nil)
    (t ;trimming the event list
      (let ((explicable-setters
              (accumulate '(lambda (x) (car x))
                          (mapcar '(lambda (x) (cons
                                                (find-printing-function x)
                                                x))
                                (cond ((boundp '*starters*) *starters*))))))
          (fork-list nil)
          (join-list nil))
        ;now rifle down and find forks and joins
        (do ((evs (cdr *events*) (cdr evs))
              (event (car *events*) (car evs))
              (last *starters* (ncons event)))
              ((null event) nil)
              ;format of fork list is exactly as before
              (cond ((eq (car event) 'FORK)
                     (push (cons last (cdr event))
                           fork-list))
                    ((eq (car event) 'JOIN)
                     (push (cons (car evs) (cdr event))
                           join-list))))
          ;now insert punctuation and fork/join entries
          (do ((nn (cdr *events*) (cdr nn))
                (event (car *events*) (car nn))
                (so-far (cond (explicable-setters
                              (cons 'END-STARTERS
                                    (mapcar 'cdr explicable-setters)))
                              (t (list 'END-STARTERS *focus*))))))
              ((null event)
               (nreverse so-far))

              (caseq (car event)
                (PATH-END
                 (cond ((equal (cadr so-far) 'AND-WHEN)
                        ;zero-length path
                        (setq so-far (cddr so-far)))
                      ((equal (car so-far) 'PATH-END))
                      (t (push 'PATH-END so-far))))
                (DONE
                 (cond ((equal (car so-far) 'DONE))
                       (t (push 'DONE so-far))))
                ((FORK JOIN) nil);already processed
                (t ;the hard part
                 (let* ((cell (car event))
                       (rule (cdr event))
                       (cell-printer (find-printing-function cell)))

```

```

(rule-printer (find-printing-function rule))
(fork (lookup-fork event fork-list))
(join (lookup-join event join-list)))

;;;formatting of an element
;rule must print in form of
;a statement about the cause of a value, related
;to the physical part model it comes from.
(let ((other-causes
      (accumulate '(lambda (x)
                     (find-printing-function x))
                   (delete (car (last-event so-far))
                           (delete event (fresh-copy join))))))
      (other-reference
        (cond ((or (equal (last-event so-far)
                          (car fork))
                  (unprintable? (car fork))) nil)
              (t (car fork)))))
      :prefer joins to forks
      (cond ((and other-reference
                  (member (car other-reference)
                          other-causes))
              (setq other-reference nil)))
      (cond (other-reference
              (push 'AND-WHEN so-far)
              (push other-reference so-far)))
      (cond (other-causes
              (push 'AND-ALSO so-far)
              (setq so-far
                    (append (nreverse other-causes)
                            so-far))
              (push 'END-AND-ALSO so-far)))
      (cond ((unprintable? event))
            (t (push event so-far))))))

(defun unprintable? (x)
  (cond ((atom x)
        (not (find-printing-function x)))
        (t (and (null (find-printing-function (car x)))
                 (null (find-printing-function (cdr x)))))))

(defun last-event (things) ;first thing that is not an atom
  (cond ((null things) nil)
        ((atom (car things)) (last-event (cdr things)))
        (t (car things))))

```

;;;Turn cleaned up event list into a big set of strings

```
(defun transform-events-to-strings (*cleaned-events* event-name graph-name)
  (cond ((not (boundp '*cleaned-events*)))
    (t ;precondition around
```

;;;The event list can be described as a finite state

;;;grammar, but needs some fancy stuff to get punctuation right.

```
(let ((events *CLEANED-EVENTS*)
      (EVENT-STRING (list (ncons "When") (ncons " "))))
  (graph-rule nil)
  (graph-cell nil))
(setq *graphics-events-accumulator* nil)
;;add start conditions
(do ((start-guy (car events) (car events))
    (starters nil))
  ((or (eq (car events) 'END-STARTERS)
       (null events))
   (setq event-string (append (Conjunctive-if-needed
                               (mapcar 'make-event-string
                                         starters))
                              event-string)
         events (cdr events)))
  (push start-guy starters)
  (setq events (cdr events)))
(setq event-string (cons (ncons ". ") event-string))
(do ((current (car events) (car events))
    (within-and-when? nil))
  ((or (null events)
       (eq current 'DONE))
   (cond ((string-equal (caar event-string) "."))
         (t (push (ncons ". ") event-string)))
   (set event-name (nreverse (apply 'nconc event-string)))
   (set graph-name (nreverse *graphics-events-accumulator*))
   (makunbound '*graphics-events-accumulator*))
  (cond ((atom current):must be a key word
        (caseq current
          (PATH-END
           (setq event-string (cons (ncons ". ") event-string)))
          (AND-WHEN ;can only happen from a path end
           (push (ncons "
event-string)
           (push (ncons "
") event-string)
           (push (ncons "Also when") event-string)
           (push (make-event-string (cadr events))
               event-string)
           (cond ((eq (caddr events) 'AND-ALSO)
                  (setq within-and-when? t))
                 (t (setq within-and-when? nil)))
           (setq events (cdr events)))
          (AND-ALSO
           (cond (within-and-when?
                  (setq within-and-when? nil)
                  (push (ncons ". ") event-string)
                  (push (ncons "along with") event-string))
                 (t (push (ncons ". ") event-string)
                     (push (list "This" ". " "along with")
                           event-string)))
           (setq within-and-also? t))
          (END-AND-ALSO
           (setq within-and-also? nil)
           (push (ncons "causes") event-string))
          (t (cond (within-and-also?
                     (push (make-event-string current)
                           event-string))
                   (t (say $ current unrecognized keyword)
                      (break transform-events-to-strings t))))))
    (setq events (cdr events)))
```

```

(t ;now we have some sequence of connectives or
;rules. For now, just commify them.
(do ((new (car events) (car events))
    (chain nil))
    ((memq new '(PATH-END DONE AND-WHEN AND-ALSO END-AND-ALSO))
     (setq event-string
            (nconc (conjunctive-if-needed
                    (mapcar 'make-event-string
                            (nreverse chain)))
                    event-string)))
    (push new chain)
    (setq events (cdr events)))))))))

```

```

(defun make-event-string (thing)
  (let* ((functional nil)
        (res nil)
        (escape-string nil)
        (graph-rule (and (not (atom thing))
                          (find-graphics-function (cdr thing)))))
    (graph-cell (cond ((atom thing) (find-graphics-function thing))
                      (t (find-graphics-function (car thing)))))
    (cond (graph-rule
          (push graph-rule *graphics-events-accumulator*)
          (push "ESC" escape-string))
          (cond (graph-cell
                (push graph-cell *graphics-events-accumulator*)
                (push "ESC" escape-string))
              (cond ((atom thing)
                    (setq functional (find-printing-function thing))
                    (setq res
                          (cond (functional (ncons (funcall functional thing))
                                (t '("")))))
                    (t (setq functional (find-printing-function (cdr thing))
                          (cond (functional (setq res (ncons (funcall functional
                                                                (cdr thing))))
                                (setq functional (find-printing-function (car thing))
                                      (cond (functional (setq res (cons (funcall functional
                                                                (car thing))
                                                                res)))
                                ((null res) (ncons '(""))))))
                    (nconc escape-string (nreverse res))))
              (nconc escape-string (nreverse res))))))

```

```

(defun conjunctive-if-needed (l1)
  (setq l1 (nreverse l1))
  (cond ((null l1) nil)
        ((= (length l1) 1) l1)
        ((= (length l1) 2.) (list (car l1) (ncons "and") (cadr l1)))
        (t (do ((l (cdr l1) (cdr l))
                (res (ncons (car l))))
            ((null l)
             (setq res (nreverse res))
             (cons (car res)
                   (cons (ncons "and") (cddr res))))
            (push (ncons ", ") res)
            (push (car l) res)
            ))))

```

;;;To specialize the description for feedback, must  
 ;;prune out everything that doesn't happen in the loop.  
 ;;We first get all the paths, and then return a new event list

```
(defun prune-non-loop-events (events)
  (cond ((or (not (boundp '*focus*))
             (not (boundp '*challenger*))) events)
        (t
         (let ((in-loop-rules
                 (scratch
                  (apply 'append
                         (append
                          (mapcar 'cdr (paths-to-target *challenger*
                                                         *focus*))
                          (mapcar 'cdr (paths-to-target (sets? (informant?
                                                             *challenger*))
                                                         *focus*)))))
               (accumulate '(lambda (event)
                              (or (memq (car event)
                                          '(END-STARTERS
                                            FORK
                                            JOIN
                                            AND-WHEN
                                            PATH-END
                                            DONE
                                            AND-ALSO
                                            END-AND-ALSO))
                                  (member (cdr event)
                                          in-loop-rules)))
                            events))))))
```

;;;finding paths for the clash

```
(defun paths-to-clash (focus)
  (setq *clash-paths*
        (apply 'append (mapcar '(lambda (x)
                                   (paths-to-target x focus))
                                *disputants*))))
```

```
(defun paths-to-target (cell target)
  (cond ((or (not (cellp? cell))
             (not (cellp? target))
             (not (known? cell))
             (not (known? target))) nil)
        (t (setq *paths-to-target* nil);accumulate globally
            (walk-to-target cell target nil)
            (mapcar '(lambda (x) (cons cell x)) :to handle disputants case!
                    *paths-to-target*))))
```

```
(defun walk-to-target (cell target path)
  (cond ((or (null cell) :pessimist!
             (null target)))
        ((eq cell target) (push path *paths-to-target*)))
  (t (let ((informer (informant? cell)))
      (cond ((or (atom informer)
                  (structural-assumption? informer)))
            (t (setq path (cons informer path))
                (mapc '(lambda (cell)
                        (walk-to-target cell target path))
                      (uses? informer))))))
```

```
;;;EXPLAIN
;;;print in reasonable fashion a list of strings

(defun explain-the-events (event-string graphics-list)
  (cond ((or (not (boundp '*using-aed*))
             (null *using-AED*)))
        (terpri)
        (pretty-print-list-of-strings event-string)
        (makunbound '*graphics-queue*))
    (t :print into AED area
        (clear-graphics-window)
        (draw-device)
        (clear-text-window)
        (setq *graphics-queue* graphics-list)
        (pretty-print-to-AED event-string)
        (setq *graphics-queue* nil))))))

(defun pop-graphics-queue ()
  (cond ((and (boundp '*graphics-queue*)
              *graphics-queue*)
        (let ((graphics-event (car *graphics-queue*)))
          (eval graphics-event))
        (setq *graphics-queue* (cdr *graphics-queue*))))))

(setq *print-escape-code* 'pop-graphics-queue)
```

```

:--LISP--
::;Part-Oriented graphics code for AED
::
::Internal coordinates are integers, which are
::converted to drawing coordinates on the fly.
::
::Globals
::AEDDRAW:PARTS - what is to be drawn
::AEDDRAW:SCALE
::*CURRENT-DRAW-ALIST* - Part specs as they should be drawn
::*CURRENT-COLOR-ALIST* - Color to draw them in

(includef '(:lstead) aed dcls))
(includef '(:lstead) gl dcls))
(includef '(:lstead) turtle dcls))

(DECLARE (SPECIAL aeddraw:parts
                 aeddraw:scale
                 *current-draw-alist*
                 *current-color-alist*
                 *blinker-on-time*
                 *blinker-off-time*
                 screen:x-offset
                 screen:y-offset
                 screen:scale)
  (NOTYPE aeddraw:parts
    *current-draw-alist*
    *current-color-alist*
    (aeddraw:draw-parts notype)
    (aeddraw:erase-parts notype)
    (parse-and-draw notype fixnum)
    (fill-angled-box flonum flonum flonum flonum flonum)
    (fill-angled-box-middle flonum flonum flonum flonum flonum)
    (aeddraw:add-to-alist notype notype notype)
    (aeddraw:change-color notype fixnum)
    (aeddraw:draw-part notype)
    (aeddraw:erase-part notype)
    (splice-parts-into-alist notype notype)
    (draw-part-blinker notype)
    (aeddraw:blink-and-change-part-color notype fixnum))
  (FLONUM screen:x-offset
    screen:y-offset
    screen:scale
    (fx-from-coords notype flonum)
    (fy-from-coords notype flonum)
    (fdx-from-coords notype flonum)
    (fdy-from-coords notype flonum)
    (aeddraw:convert-length flonum))
  (FIXNUM *blinker-on-time*
    *blinker-off-time*
    (color-bars flonum flonum)))

```

```

:::Basic Part drawing code
::The parts list of a device is assumed to be an alist
::whose keys are the names of parts and whose items
::are an alist containing DRAW and COLOR specifications.
::When told to draw a parts list, the parts in it are
::drawn and the current DRAW and COLOR alists are updated.
::Should the picture need to be refreshed, these lists
::are used.
:::The parts lists can be PARTIAL - this is how "motion" is
::depicted in the current system. The parts involved are erased
::and then redrawn in a new position.

(defun aeddraw:draw-parts (parts)
  ;this code initializes the stuff
  (setq *CURRENT-COLOR-ALIST* NIL
        *CURRENT-DRAW-ALIST* NIL)
  (do ((pp (cdr parts) (cdr pp))
      (part (car parts) (car pp))
      (draw nil)
      (color nil)
      (draw-color 0))
      ((null part) )
    (setq draw (cdr (assoc 'DRAW (cdr part)))
          color (cdr (assoc 'COLOR (cdr part))))
    (cond ((null draw)
          (setq draw (cdr (assoc (car part) *CURRENT-DRAW-ALIST*))))
          (t (aeddraw:add-to-alist (car part) draw '*CURRENT-DRAW-ALIST*)))
    (cond (draw
          (cond ((null color)
                (setq color (cdr (assoc (car part) *CURRENT-COLOR-ALIST*)))
                (cond (color (setq draw-color color))
                      (t (setq draw-color background-color))))
                (t (setq draw-color (eval color))
                    (aeddraw:add-to-alist (car part) draw-color
                                           '*CURRENT-COLOR-ALIST*)))
          (cond (debug-valvepic
                (say part is $ (car part))
                (break valvepic t)))
          (mapc '(lambda (spec)
                  (parse-and-draw spec draw-color))
                draw))))))

(defun aeddraw:draw-part (part)
  (let* ((draw (cdr (assoc part *CURRENT-DRAW-ALIST*)))
        (color (cdr (assoc part *current-color-alist*)))
        (cond (draw
                (mapc '(lambda (spec)
                        (parse-and-draw spec color))
                      draw))))))

(defun aeddraw:erase-part (part)
  (let* ((draw (cdr (assoc part *CURRENT-DRAW-ALIST*)))
        (color screen-color))
    (cond (draw
          (mapc '(lambda (spec)
                  (parse-and-draw spec color))
                draw))))))

(defun aeddraw:erase-parts (parts)
  (do ((pp (cdr (reverse parts)) (cdr pp))
      (part (car (reverse parts)) (car pp))
      (draw nil)
      (color nil)
      (draw-color 0))
      ((null part) )
    (setq draw (cdr (assoc (car part) *CURRENT-DRAW-ALIST*))
          color (cdr (assoc (car part) *CURRENT-COLOR-ALIST*)))
    (cond (draw
          (cond ((null color)
                (setq color (cdr (assoc (car part) *CURRENT-COLOR-ALIST*)))
                (cond (color (setq draw-color color))
                      (t (setq draw-color background-color))))
                (t (setq draw-color (eval color))
                    (aeddraw:add-to-alist (car part) draw-color
                                           '*CURRENT-COLOR-ALIST*)))
          (cond (debug-valvepic
                (say part is $ (car part))
                (break valvepic t)))
          (mapc '(lambda (spec)
                  (parse-and-draw spec draw-color))
                draw))))))

```

```
(setq draw-color background-color)
(cond (debug-valvepic
      (say part is $ (car part))
      (break valvepic t)))
(mapc '(lambda (spec)
        (parse-and-draw spec draw-color))
      draw))))
```

:::hacking parts

```
(defun aeddraw:ADD-TO-ALIST (name val ll)
  (cond ((not (boundp ll))
    (set ll (list (cons name val))))
    (t (let ((current (assoc name (symeval ll))))
      (cond (current (rplacd current val))
        (t (set ll
          (cons (cons name val)
            (symeval ll))))))))))

(defun aeddraw:change-color (part newcolor)
  (let* ((part-alist (cdr (assoc part aeddraw:parts)))
    (draw (cdr (assoc 'draw part-alist)))
    (current-color (assoc part *CURRENT-COLOR-ALIST*)))
    (cond (draw
      (mapc '(lambda (spec)
        (parse-and-draw spec newcolor))
        draw)))
    (aeddraw:add-to-alist (car part) newcolor *CURRENT-COLOR-ALIST*)))
```

```
(defun AEDDRAW:Change-Part-Color (partname color)
  ;assumes the part does not obscure anything else
  (aeddraw:add-to-alist partname color *CURRENT-COLOR-ALIST*)
  (aeddraw:draw-part partname))
```

```

(defun splice-parts-into-alist (newparts alist)
  (do ((pieces (cdr newparts) (cdr pieces))
      (piece (car newparts) (car pieces))
      (alist-piece nil)
      (alist-value (syneval alist)))
    ((null piece) (set alist alist-value))
    (setq alist-piece (assoc (car piece) alist-value))
    :such as MainValve
    (cond ((null alist-piece)
      :new thing
      (setq alist-value (cons piece alist-value)))
      (t ;otherwise put in the new values
      (mapc '(lambda (stuff)
        (let ((local-stuff (assoc (car stuff)
                                   (cdr alist-piece))))
          (cond (local-stuff
            (rplacd local-stuff
              (cdr stuff)))
            (t (rplacd alist-piece
              (cons stuff
                (cdr alist-piece)))))))
        (cdr piece))))))

```

;;Drawing primitives  
 ;;Without a graphics editor, defining the graphic depiction  
 ;;of a device can be performed by laying it out on graph  
 ;;paper and breaking it up into boxes and triangles.

```
(defun parse-and-draw (spec color)
  (aed:sec color)
  (caseq (car spec)
    (BOX ;draw a filled in box
      (let* ((l1 (cdr spec))
              (fx (fx-from-coords l1 aeddraw:scale))
              (fy (fy-from-coords l1 aeddraw:scale))
              (fdx (fdx-from-coords l1 aeddraw:scale))
              (fdy (fdy-from-coords l1 aeddraw:scale)))
        (gl:mov fx fy)
        (gl:solidboxrel fdx fdy)))
    (TRIANGLE ;draw a filled triangle
      (let* ((l1 (cddr spec))
              (fx (fx-from-coords l1 aeddraw:scale))
              (fy (fy-from-coords l1 aeddraw:scale))
              (fdx (fdx-from-coords l1 aeddraw:scale))
              (fdy (fdy-from-coords l1 aeddraw:scale)))
        (penup)
        (setxy-simple fx fy)
        (pendown)
        (setxy-simple (+$ fx fdx) (+$ fy fdy))
        (caseq (cadr spec) ; need to use turtle graphics
          (1 (setheading 180.0)
              (forward-simple fdy)
              (setheading 270.0)
              (forward-simple fdx)
              (penup)
              (setxy-simple (+$ fx (//$ fdx 2.0))
                            (+$ fy (*$ fdy 0.8)))
              (fillregion-simple))
          (2 (setheading 270.0)
              (forward-simple fdy)
              (setheading 180.0)
              (forward-simple fdx)
              (penup)
              (setxy-simple (+$ fx (//$ fdx 2.0))
                            (+$ fy (*$ fdy 0.2)))
              (fillregion-simple))
          (3 (setheading 180.0)
              (forward-simple fdx)
              (setheading 90.0)
              (forward-simple (abs fdy))
              (penup)
              (setxy-simple (+$ fx (//$ fdx 2.0))
                            (+$ fy (*$ fdy 0.8)))
              (fillregion-simple))
          (4 (setheading 90.0)
              (forward-simple (abs fdy))
              (setheading 180.0)
              (forward-simple fdx)
              (penup)
              (setxy-simple (+$ fx (//$ fdx 2.0))
                            (+$ fy (*$ fdy 0.2)))
              (fillregion-simple))
          (t (penup)))
        (penup)))
    (ANGLE-BOX
      (let* ((l1 (cdr spec))
              (fx (fx-from-coords l1 aeddraw:scale))
              (fy (fy-from-coords l1 aeddraw:scale))
              (orient (caddr l1)))
```

```
(len (aeddrow:convert-length (caddr 1)))  
(wid (aeddrow:convert-length (car (caddr 1))))  
(fill-angled-box fx fy orient len wid)))  
(ANGLE-BOX-middle  
  (let* ((11 (cdr spec))  
         (fx (fx-from-coords 11 aeddrow:scale))  
         (fy (fy-from-coords 11 aeddrow:scale))  
         (orient (caddr 1))  
         (len (aeddrow:convert-length (caddr 1)))  
         (wid (aeddrow:convert-length (car (caddr 1))))  
         (fill-angled-box-middle fx fy orient len wid)))  
    (t nil)))
```

```
(defun fx-from-coords (c1 scale)
  (+$ SCREEN:X-OFFSET
    (*$ (//$ (float (car c1)) scale)
      SCREEN:SCALE)))

(defun fy-from-coords (c1 scale)
  (+$ SCREEN:Y-OFFSET
    (*$ (//$ (float (cadr c1)) scale)
      SCREEN:SCALE)))

(defun fdx-from-coords (c1 scale)
  (*$ (//$ (-$ (float (caddr c1)) (float (car c1))) scale)
    SCREEN:SCALE))

(defun fdy-from-coords (c1 scale)
  (*$ (//$ (-$ (float (caddr c1)) (float (cadr c1))) scale)
    SCREEN:SCALE))

(defun aeddraw:convert-length (num)
  (*$ (//$ (float num) aeddraw:scale)
    SCREEN:SCALE))
```

;;;More turtle graphics

```
(defun fill-angled-box (fx fy orient len wid)
  (penup)
  (setxy-simple fx fy)
  (setheading orient)
  (pendown)
  (forward-simple len)
  (left -90.0)
  (forward-simple wid)
  (left -90.0)
  (forward-simple len)
  (left -90.0)
  (forward-simple wid)
  (penup)
  (let ((radians (deg-rad orient))
        (other (deg-rad (-$ orient 90.0)))))
    (setxy-simple (+$ fx
                     (*$ len 0.5 (cos radians))
                     (*$ wid 0.5 (sin radians)))
                  (+$ fy
                     (*$ len 0.5 (sin radians))
                     (*$ wid -0.5 (cos radians)))))
  (fillregion-simple))

(defun fill-angled-box-middle (midx midy orient len wid)
  (let* ((radians (deg-rad orient))
         (fx (-$ midx
                (*$ len 0.5 (cos radians))
                (*$ wid 0.5 (sin radians)))
          (fy (-$ midy
                (*$ len 0.5 (sin radians))
                (*$ wid -0.5 (cos radians)))))
    (penup)
    (setxy-simple fx fy)
    (setheading orient)
    (pendown)
    (forward-simple len)
    (left -90.0)
    (forward-simple wid)
    (left -90.0)
    (forward-simple len)
    (left -90.0)
    (forward-simple wid)
    (penup)
    (setxy-simple midx midy)
    (fillregion-simple)))
```

```
:::blinking system
```

```
(setq *blinker-on-time* 15
      *blinker-off-time* 15)
```

```
(defun draw-part-blinking (part)
  (let* ((current-color (cdr (assoc part *current-color-alist*)))
         (blinking-color (+ 8. current-color)))
    (aeddrow:change-part-color part blinking-color)
    (aed:sbl blinking-color 0 0 0 *blinker-on-time* *blinker-off-time*)
    (tyi)
    (aed:sbl blinking-color 0 0 0 *blinker-on-time* 0)
    (aeddrow:change-part-color part current-color)))
```

```
(defun aeddrow:blink-and-change-part-color (part newcolor)
  (let* ((blinking-color (+ 8. newcolor)))
    (aeddrow:change-part-color part blinking-color)
    (aed:sbl blinking-color 0 0 0 *blinker-on-time* *blinker-off-time*)
    (tyi)
    (aed:sbl blinking-color 0 0 0 *blinker-on-time* 0)
    (aeddrow:change-part-color part newcolor)))
```

```
(defun aed:blinker-cm (cm)
  ;; Set a color matrix to the AED default.
  (fillarray cm
    '(0 0 0 ; Black.
      255 0 0 ; Red.
      0 255 0 ; Green.
      255 255 0 ; Yellow.
      0 0 255 ; Blue.
      255 0 255 ; Magenta.
      0 255 255 ; Cyan.
      255 255 255 ; White.
      0 0 0 ;Blinking Black.
      255 0 0 ;Blinking Red.
      0 255 0 ;Blinking Green.
      255 255 0 ;Blinking Yellow.
      0 0 255 ;Blinking Blue.
      255 0 255 ;Blinking Magenta.
      0 255 255 ;Blinking Cyan.
      255 255 255 ;Blinking White.
      0 ; Rest to black
    ))
  t)
```

```
(defun start-blinker ()
  (aed:blinker-cm aed:ct)
  (aed:sct aed:ct 8. 8.))
```

```
;;;Put up color bars on AED
(defun color-bars (xbot ybot)
  (let ((old text-color))
    (do ((i 0 (1+ i))
        (y ybot (+$ y *char-height* *line-feed-height*)))
      ((> i 7)
       (setq text-color old))
      (setq text-color i)
      (aed:sec i)
      (gl:mov xbot y)
      (gl:solidbox *char-height* *char-height*)
      (setq *text-x* (+$ xbot (*$ 1.5 *char-height*))
            *text-y* y)
      (princ-aed i))))
```

```
;-*-LISP*-  
:  
::: AED interface code for Feedback Trainer  
::  
  
:: This collection of programs makes the AED appear to be  
:: divided into two windows, one for text and the other for  
:: graphics. The graphics window is on top. The size of the  
:: windows are specified by the variables  
:: SCREEN:<GRAPHICS or TEXT>-<LEFT, WIDTH, HEIGHT, or LOWER>  
::  
:: Text is placed on the AED when *AED-ECHO* is T  
::  
:: The parameters for AED text are  
::: *AED-LINEL* - Number of characters in a line  
::: *CHAR-HEIGHT* - Height of character in AED units  
::: *CHAR-WIDTH* - Width of character in AED units  
::: *LINE-FEED-HEIGHT* - distance between lines  
::: *AED-NUMBER-OF-LINES* - Number of lines in text area  
::: TEXT-COLOR - Color to print text in  
  
; default value  
  
(setq text-color 7)  
  
(setq *char-height* (//$ 1.0 64.0)  
  *char-width* (//$ 1.0 64.0)) ;BRoberts' measurements  
  
(setq *line-feed-height* (//$ 1.0 40.0))  
  
(setq *AED-Number-of-lines* 20.)  
  
(setq screen:text-left 0.0  
  screen:text-lower 0.0  
  screen:text-height 0.5  
  screen:text-width 1.0  
  screen:graphics-left 0.0  
  screen:graphics-lower 0.5  
  screen:graphics-height 1.0  
  screen:graphics-width 1.0)  
  
(setq screen:text-home-x 0.0  
  screen:text-home-y (-$ screen:text-height *char-height*)  
  screen:graphics-home-x 0.5  
  screen:graphics-home-y 0.5)  
  
(setq *AED-linel* 55.)
```

```

(defun pretty-print-to-AED (strings)
  (let ((line-length *aed-line-length*))
    (and *using-AED* (aed:sec text-color)))
    (do ((str (cdr strings) (cdr str))
          (this (wordify (car strings)) (wordify (car str)))
          (line-left line-length) Amount of char space left on line
          (stringpos 0)
          (stringlen 0)
          (last-word ""))
        ((null str))
        (do ((words this (cdr words))
              (word "")
              (len 0))
            ((null words))
            :first take action on the last word
            (setq word (car words))
              len (string-length word))
          (cond ((string-equal word "&&&ESCAPE&&&")
                  :don't print-this is a signal!
                  (and (boundp '*Print-escape-code*)
                       (funcall *print-escape-code*)))
                (t
                 (cond ((string-equal "." last-word) (princ-AED " ")
                       (setq line-left (1- line-left)))
                       ((string-equal "." last-word) (princ-AED " ")
                       (setq line-left (~ line-left 2)))
                       ((not (or (string-equal word ".")
                                (string-equal word ".")
                                (string-equal word ";")
                                (newline? last-word)))
                        :put a space in between words
                        (princ-AED " ")
                        (setq line-left (1- line-left))))
                     (cond ((newline? last-word)
                             (new-AED-line)
                             (setq line-left line-length)))
                     (cond ((and (not (punctuation? word))
                                (> len line-left))
                             (new-AED-line)
                             (setq line-left line-length)))
                     (cond ((or (and (string-equal last-word ".")
                                     (string-equal word "."))
                               (and (string-equal word "
")
                                   (= line-left line-length))))
                             (t (princ-AED (capitalize? word)
                                     (string-equal last-word "."))
                               (setq line-left (- line-left len))
                               (setq last-word word)
                               ))))))))

(defun clear-text-window ()
  (cond (*AED-ECHO*
        (aed:sec screen-color)
        (gl:mov screen:text-left screen:text-lower)
        (gl:solidbox screen:text-width screen:text-height)
        (setq *text-x* screen:text-home-x
              *text-y* screen:text-home-y
              *aed-line-index* *aed-number-of-lines*)
        (gl:mov screen:text-home-x screen:text-home-y);home cursor to top of area
        )))

(defun clear-graphics-window ()
  (cond (*using-AED*
        (aed:sec screen-color)
        (gl:mov screen:graphics-left screen:graphics-lower)

```

```
(gl:solidbox screen:graphics-width screen:graphics-height)
(gl:mov screen:graphics-home-x screen:graphics-home-y);home cursor to top of area
)))
```

```
(defun initialize-text-window () (clear-text-window))
```

```
(defun initialize-graphics-window () (clear-graphics-window))
```

```
(defun new-AED-line ()
  (cond (*AED-ECHO*
        (cond ((not (boundp '*AED-LINE-INDEX*))
              (setq *AED-LINE-INDEX* *AED-NUMBER-OF-LINES*))
              ((< *AED-line-index* 1)
               (princ "AED More?")
               (tyi-a-y)
               (clear-text-window)
               (gl:mov screen:text-home-x screen:text-home-y)
               (setq *aed-line-index* *aed-number-of-lines*))
              (*using-aed*
               (setq *text-y* (max 0.0 (-$ *text-y* *line-feed-height*))
                     *text-x* screen:text-home-x)
               (gl:mov *text-x* *text-y*)
               (setq *aed-line-index* (1- *aed-line-index*)))))))
  (cond (*TTY-ECHO* (terpri))))
```

```
(defun tyi-a-y ()
  (do ((thing (readch) (readch)))
      ((or (eq thing '|Y|)
           (eq thing '|y|)))))
```

```
(defun princ-AED (thing)
  (cond (*AED-ECHO*
        (aed:sec text-color)
        (gl:mov *text-x* *text-y*)
        (aed:tmode)
        (princ thing aed:out)
        (gl:rcp) :read-cursor-position
        (setq *text-x* gl:rcp-x
              *text-y* gl:rcp-y)))
  (cond (*TTY-ECHO* (princ thing))))
```

```
(defun ECHO-AED (thing)
  (cond (*AED-ECHO*
        (aed:sec text-color)
        (gl:mov *text-x* *text-y*)
        (aed:tmode)
        (princ thing aed:out)
        (gl:rcp) :read-cursor-position
        (setq *text-x* gl:rcp-x
              *text-y* gl:rcp-y))))
```